Implementing Prato, a database on demand service (*extended abstract*)

Soila Pertet, John Wilkes, and Jay Wylie

HP Labs

soila@cmu.edu, john.wilkes@hp.com, jay.wylie@hp.com

Configuring a database is much easier than it used to be, but it's still a chore that many of us would rather not perform. The investment in hardware and people necessary to make a large, powerful database available is prohibitive for short-term tasks. As a result, applications that might use a database do without; queries take much longer than they need to; useful business analytics is not performed; and people waste time learning how to make the database fast, rather than focusing on their primary business function.

Prato solves these problems by offering customers a private, virtual, DBMS appliance that can be sized up to several hundred nodes, and made available on demand, in minutes. The initial Prato prototype is built around a main-memory DBMS on which decision support queries run an order of magnitude faster than on a traditional DBMS. Future versions will make the database resilient to a wide range of failures; completely self-managing; and capable of supporting multiple back-end database types.

The research problems are centered on how to automate Prato's control system in the face of a large-scale distributed system that has many failure-prone components, supports multiple users with varying degrees of sophistication and demands, and has to handle events in a completely lights-out manner. Towards this end, we employ a policy-based approach to the design and implementation of Prato's control system.

1. Introduction

The end-user visible goal of Prato is to make private, virtual, DBMS appliances available on demand. To accomplish this, we apply service-oriented architecture (SOA) techniques and self-managing systems research to the problem of database provisioning. We have built an initial prototype that is already being used by internal customers in a limited manner. We are using early customer experiences to validate the use cases and design of Prato. We also expect such customer experience to enable us to identify the most valuable additional features and services to pursue next.

There are many data warehouse vendors (e.g., Teradata, Netezza, Oracle, and now HP with NeoView). Such vendors sell database appliances or software for exclusive use by an end customer. The Prato service differs in that it can support multiple (potentially competing) customers on a shared infrastructure, via a pay-per-use service model. These differences make the power of data warehouses available to customers who heretofore could not afford the necessary capital investment. It also allows the individual instances to be flexed and resized as needed. What is a simple reconfiguration for Prato turns into a fork-lift upgrade with a traditional database appliance. (We sometimes refer to Prato as a *virtual* DBMS-appliance provider for this.)

The research goal of Prato is to learn how to deliver this end-user experience from a completely automated system that is capable of handling user requests and failures without human intervention. We want to change the provisioning of information services from a manual task to a fully automatic one. This means that we are interested in:

1. How best to capture customer needs, without dictating the way those needs are addressed: in our case, how much database is needed, and how important it is that it stay available.
2. Automatically translating such needs into implementation choices, and selecting between different designs: in our case, what kind of DBMS options to set, and how much to invest in different failure-recovery alternatives.
3. Completely automating the management of the Prato service provider: lights-out self-management is the end goal, even in the face of conflicting user requirements, and failures.
4. Service composition: how Prato interacts with other service providers.

We discuss each of these topics in this paper, but emphasize our policy-based approach to automating the management of the Prato service provider.

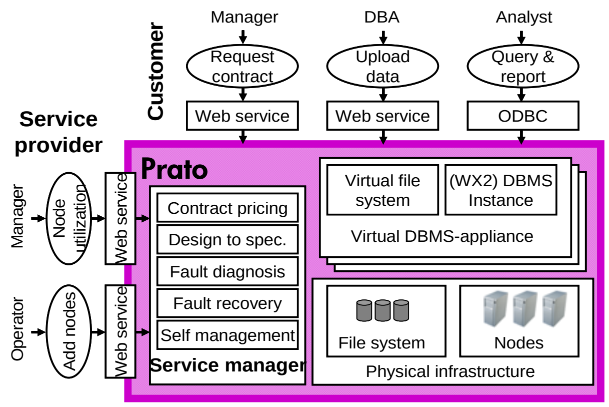

Figure 1: Prato service provider architecture.

2. Prato service provider

The Prato service provider has three main parts (shown in Figure 1):

1. A database engine that is used to create DBMS appliance instances.
2. A resource pool, which provides processors for the DBMS appliances and file storage space for user data, such as internal backups/snapshots.
3. The service manager, a control system, which is in charge of orchestrating all the other parts, responding to requests for new DBMS appliances, and recovering from failures.

2.1. Database engine

Since the Prato project is ultimately about controlling database management systems, not building them, it is constructed on top of back-end database systems that we obtain from others.

The first of these is WX2, from Kognitio [Kognitio2007]. We chose it because it has two interesting properties: first, it performs very fast scans of in-memory data, which can often obviate the need for indexes, thereby easing the process of adding new data into the system. Second, and perhaps most important, it aggregates multiple Linux compute nodes into a single DBMS instance – that is, a query can execute on multiple nodes at a time. This means that a single query can bring large amounts of main memory to bear on a problem – in our case, up to about 1TB. Because WX2 was designed from the ground up to be a distributed system, it has proven a natural partner and target for our distributed control system research objectives. In addition, because the number of nodes that a single WX2 instance runs on can be controlled, initial system sizing and flexing to add additional resources is relatively straightforward. This was a good match to our desire to provide tailored, per-customer DBMS appliances.

Kognitio, our DBMS provider, is in the same business as Prato: they recently announced a "Data Warehouse Challenge" in which they promise to "build you a Data Warehouse in just 14 days for you to analyze your data." Kognitio relies on their internal consultants and experts to make such an endeavor feasible. By comparison, Prato is designed to accomplish a similar task without any manual intervention. Besides the potential cost savings on the service provider side, this can lead to more rapid DBMS availability for its customers: minutes for database provisioning and tens of minutes for data import.

Our future plans call for us to support additional DBMS back ends – in particular, the open-source MySQL, followed by a control interface to an HP NeoView platform [NeoView2007]. Clustered versions of MySQL have some of the same properties as WX2, although over a smaller range of sizes. They add additional design points, with a variable mixture of front-end and back-end nodes.

2.2. Resource pool

For the resource pool, we were fortunate to have access to several hundred HP DL360 nodes, which remained from the collaboration between HP and DreamWorks on *Shrek 2* and *Madagascar* [Beckett2004]. Without such a pool, we would never have learned some of the more interesting (and frustrating) lessons about controlling large-scale systems. In addition, some of our colleagues made available a few terabytes of file server space that we use to hold a few large datasets.

Currently, Prato takes physical possession of the nodes in the resource pool that it allocates to different database appliances. We are working to make Prato a client of a physical resource provider service – one that dynamically rents out physical nodes to hosted services such as Prato, on an on-demand basis.

2.3. Prato service manager

We believe that the most novel part of Prato is the service manager, the control system that we are building around the other parts (see Figure 1). The Prato service manager orchestrates all of the operations performed by the Prato service: it is responsible for designing virtual DBMS-appliances that meet client contract requests, pricing such contracts, allocating physical resources for such contracts, instantiating a specific virtual DBMS-appliance on allocated resources for accepted contracts, diagnosing failures, planning recoveries from failures, and so on. These responsibilities fall into three main categories:

1. Create/relinquish a DBMS, which is invoked by the customer in their *Manager* role.
2. Upload/download data, which also performs schema and table instantiation, which is invoked by the customer in their *Database Administrator (DBA)* role.
3. Query execution, provided via a standard ODBC feed, which is performed by the customer in their *Analyst* role.

In addition, the Prato service itself has *business-manager* and *operator* roles which the Prato service manager must support. Our goal is to make these as little needed as possible – particularly the latter.

2.4. Design and implementation

We have built the Prato service manager using Enigmatec's Execution Management System (EMS) [Enigmatec2007]. Again, this came about because we were looking for a way to bootstrap our research efforts to support a distributed, failure tolerant system. EMS offers us location-independent workflow management and a convenient engine for implementing policies, which means we can concentrate our efforts on the policies and objectives, rather than spending time developing a low-level platform to route events, and cope with local failures in the control system itself. To make our EMS system more robust, we are pairing its distributed information store (replicated across the main memory of several EMS processes on different nodes) with a stable copy of important data on persistent storage.

We have organized the Prato service manager in a hierarchy. Figure 2 shows the service organization we have implemented using EMS for the Prato service manager. At the highest level, the PratoService offers interfaces to clients to request a DBMS appliance, accept a contract offered by the PratoService in response to a DBMS appliance request, notify a client that their contracted DBMS appliance is ready, let a client release a DBMS appliance, and to notify a client that their contracted DBMS appliance has expired.

There are two sub-trees under the PratoService: the WX2Manager and the NodePoolManager. The former manages virtual WX2 DBMS appliances and the latter manages the nodes in the resource pool. Nodes from our compute cluster are managed exclusively by the NodePoolManager until they are allocated to some DBMS appliance. Once allocated, a node is managed by the DBMS appliance to which it is allocated. Once an appliance is released, the nodes bound to it return to the management of the NodePoolManager.

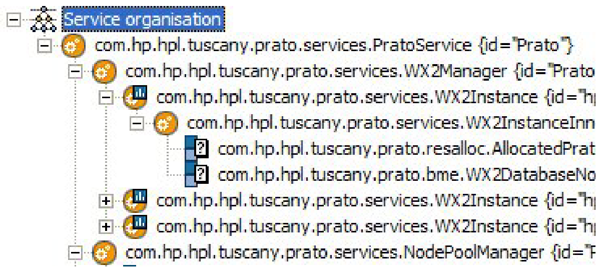
```
Service organisation
  com.hp.hpl.tuscany.prato.services.PratoService {id="Prato"}
    com.hp.hpl.tuscany.prato.services.WX2Manager {id="Prato
      com.hp.hpl.tuscany.prato.services.WX2Instance {id="h
        com.hp.hpl.tuscany.prato.services.WX2InstanceInn
          com.hp.hpl.tuscany.prato.resalloc.AllocatedPrat
          com.hp.hpl.tuscany.prato.bme.WX2DatabaseNo
      com.hp.hpl.tuscany.prato.services.WX2Instance {id="h
      com.hp.hpl.tuscany.prato.services.WX2Instance {id="h
    com.hp.hpl.tuscany.prato.services.NodePoolManager {id="P
```

Figure 2: Prato EMS *Service organization*.

The NodePoolManager implements policies for when a node is allocated to an appliance, released from the appliance, first comes up and joins the resource pool, and goes down/leaves the resource pool. It also offers an interface to query the resources available in the resource pool. Figure 3 illustrates the flow of the NodePoolManager policy for node allocation. If nodes are successfully bound to some DBMS appliance, then state about available resources is updated. If not, the lower branch of the workflow signals an error, which other policies handle.

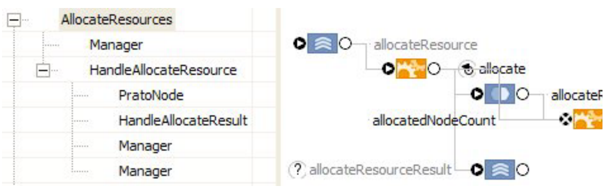

Figure 3: Node pool resource allocation.

In the example service organization shown in Figure 2, up to three WX2 DBMS appliances can be managed by the WX2Manager. Currently services cannot invoke methods on themselves in EMS. To permit self-invocation, we introduce a WX2Instance and a WX2InstanceInner: the former is a place holder that defines the policy interface for a WX2 DBMS appliance and passes events through; the latter is an internal component that responds to events.

3. Designing for self-management

We are designing the Prato service provider to run without operator intervention. This means that we need a way to delegate to it many of the decisions that are commonly made by people in a traditional service implementation. Ideally, this is best accomplished without a great deal of fine-tuning of a large number of knobs: there's little point in developing a system that is harder to control than the original – at least over the expected range of operation. Our approach to this is to pick one metric, and see how far using that metric to determine the system's behavior can take us. That metric is profit – the difference between the cost of running the Prato service and the amount it can charge its clients.

3.1. Economic approaches to self-management

A service must provide something that its customers value – or they would simply not use it. In the face of competition, a service must be exploiting an asset that its competitors cannot match if it is to remain profitable – or the competition will simply put it out of business, or customers will find a way to provide the service themselves. In our case, we intend that asset to be the automation of the service management, because we believe that a self-managing system will be cheaper to run (i.e., take fewer people), have higher responsiveness to requests (resulting in a more desirable service), be less likely to result in system outages when things go wrong (thereby reducing penalties), and rational in its allocation of resources to external needs (thereby maximizing the return on investment).

As a result, we believe that profit provides the best overall measure of the utility a service is adding to its environment. The main benefits come from having a simple, clear metric for what "goodness" means in the system. This clarity makes it easy to determine whether a choice is making things better or worse, and thus makes it a good choice for the objective function of the automated control system.

Profit has the additional benefit of getting business-people's attention. We thus try and leverage economic mechanisms and incentives to establish self-interested policies for components that overall yield self-managing behavior.

In order to drive appropriate behavior in a service, we use price signals from the service's customers to communicate how much they value different aspects of its behavior. For example, a typical contract will include both rewards for good behaviors and penalties for undesirable ones. The rewards can be as simple as a flat rate subscription (i.e., per unit time), or include utility functions over responsiveness or latency. The penalties can be as inventive as the client and service provider are willing to be. In the Prato context, we focus mainly on availability and data loss penalties.

One difficulty with using profit as the *lingua franca* of utility is that a service must be able to predict the likely cost of its actions, which is not always trivial. For example, consider how hard it is to accurately predict rare events such as failures, or future events such as tomorrow's demand for the Prato service. The former is necessary when designing and pricing a DBMS appliance for a customer, and the latter is necessary when pricing and admitting customer contracts. We believe that this is best handled explicitly as a risk management question. The good news is that the economic approach brings it clearly into the open, and provides a way to quantify it – and provides a hint that existing work in other areas, such as financial markets, could be helpful.

In a policy-driven system, the policies represent ways of recording preferred decisions, or preferred approaches to making decisions.[1] The highest-level policy in the Prato universe is to maximize profit rate – the profit per unit time, subject to a certain expected level of risk, expressed as a permitted variance in the profit-rate, together with a maximum expected loss. [2]

We can then assess other policy choices and ask "[how] does this contribute to the primary objective?" We have found that an economics-based approach to the problems of designing self-managing systems helps focus attention on those decisions that are likely to influence a service's profitability, rather than low-level mechanism choices.

In the rest of this section, we discuss how we apply economic incentives towards making two key aspects of Prato self-managing: autonomously handling client requests, and ensuring continued operation in the face of failures. We also discuss the open problem of composing self-managing components.

3.2. Automatic database instance design

Prato autonomously determines how to configure the database instance and corresponding recovery plans for a database appliance. Prato uses the requested service-level agreement (SLA) to accomplish this task.

When negotiating for a DBMS appliance from Prato, the customer proposes a preferred service level objective (SLO) and the penalties to be paid out if the SLO isn't met. Prato responds with a price. If the customer accepts it, the resulting combination becomes the customer's contract, or service level agreement (SLA). This places two burdens on Prato: it must determine how to provision and configure a database instance for the proposed contract, and it must determine how much to charge for it in order to meet its profit objective.

[1] We find it helpful to think of *policy* in the most general sense of a replaceable way of representing what to do in some situation, whether this is a goal statement or a simple condition-action rule.

[2] We could have chosen a metric such as profit per contract, or return on investment. Our analysis of risk is still at an early stage. We present it here to suggest the flavor of our approach.

Our approach to the database instance design process extends techniques first developed for storage-system availability [Keeton2004]; these use business-penalty information provided by the customer to drive the service provider decisions, while making the SLA very straightforward. For Prato, the penalties are defined as an outage penalty-rate (dollars per hour that the instance is not available to do useful work) and a data loss penalty-rate (dollars per hour that the state of the system has to be rolled backwards in order to effect a recovery).

The use of business-level penalties rather than prescriptive recipes (e.g., "you will keep at least 2 copies of this data on different media types") means that the service provider retains a great deal of freedom behind the scenes in terms of which mechanisms it should use to meet the customer needs.

In our case, the mechanisms we are designing include making local copies of customer data to provide fast recovery (we can upload a new copy in parallel from a local copy much faster than it can be downloaded from the customer data source), DBMS- and disk-level data-mirroring, hot/cold node spares, and a range of recovery procedures that offer different degrees and speeds of failure tolerance [Pertet2007].

The design emphasis in Prato is on fault tolerance. In other systems, it could well be performance. In our case, the only "performance knob" for a WX2 DBMS instance is the number of nodes allocated to it: and the main determiner of the database performance is whether or not the database will fit in memory. If it does, the result is usually such good performance that our customers do not quibble!

As we expected, we are finding that the basic principles enunciated in [Keeton2004] hold good, but the devil is in the details – for example, exactly how should we model the multitude of failure types, failure mitigation efforts, and their effects? Failures are relatively rare events, so [how] should observed failures be factored into the predictive models? When should the design of a particular instance be adjusted on the fly, and when should it be allowed to live out its life with a less-than-optimal design?

Storage failure modes and rates are much better understood then the failure modes of complete services (especially those composed of multiple services as is done in a SOA). In particular, we believe that the prior techniques need to be extended to incorporate risk tolerance aspects. It is not enough to assume a single failure event, as was done in [Keeton2004] – we need to extend this to predict the likelihood of multiple events and their consequences. We believe that explicitly including risk in the models can address the dearth of knowledge about failure modes and rates in such systems.

3.3. Policy-based self-management

Our goal is to make Prato capable of handling failure events, customer events, and all normal operation functions completely autonomously, with *no* operator intervention. This requires that we be able to recognize an event such as a failure or a request for service, determine potential courses of action, select between them, and then execute the preferred ones.

In a customer-driven event (such as a customer requesting a new database instance), the identification of the event is usually straightforward, and the state of the Prato system is likely to be reasonably well understood. The main choices are two-fold: whether it is possible (and if so, how best) to meet the request, and, if so, whether it is desirable (profitable!) to do so – i.e., admission control.

Admission control choices are determined by service-business-level policies that capture the service-provider's business choices, such as the appropriate tradeoff between the expected levels of profit and risk in a contract. Admission control decisions could also include other customer-related information such as a prior interaction history with the customer, or their reputation – e.g., whether they are likely to pay promptly! It seems that the development of such policies is still in its infancy. Prato is no exception.

Once the business-level policies have been applied, it's necessary to address more technical aspects. These include the self-management policies of each database instance and its constituent nodes. Our approach to this is to formalize the state-space through which each component can move (e.g., see Figure 4).

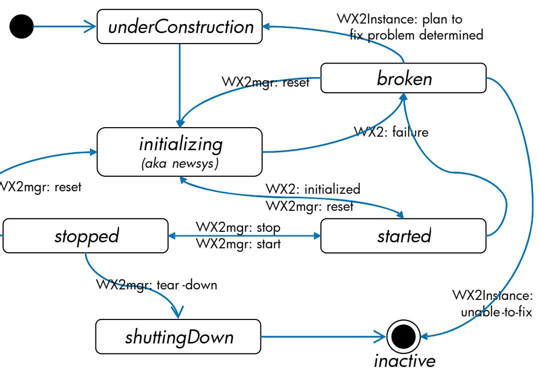

Figure 4: Prato DBMS-instance state transitions.

This state-space approach allows us to develop responses for both normal operations and for threats such as failures, and we believe that it represents a fairly powerful, general-purpose technique. Determining which transition to take when faced with a state and an event can be controlled by policies: for example, the recovery mechanism design for a database instance is itself a kind of policy.

However, one of the biggest difficulties facing responses to failures is determining exactly what state the system is in – the problem of failure detection. Part of the difficulty is simply that of gathering data about a failure event, since the failure might have taken down part of the monitoring system. Part of the difficulty is in determining what course of action is most likely to lead to a desirable outcome. Fortunately, the economics-driven approach to the SLA helps significantly here – the tradeoffs between the costs of various mechanisms and their expected return (reduced penalties) are readily understood, if these values can be predicted.

A more insidious problem is how to determine which part of the system should respond – which leads us onto the topic of composite systems and policies.

3.4. Policy composition

Like most service implementations, Prato is not internally monolithic: it is composed of the interacting, loosely-coupled components shown in Figure 2. The independence of these components presents some interesting problems, akin to ones that show up in composite services built from autonomous components.

For example, take the simple case of a node that stops responding. Who should respond? It might be a process that has run amuk, and resetting the database instance will get things working again, quickly, with relatively little loss of availability or data. Maybe the operating system has gotten itself wedged temporarily, in which case a reboot will fix things, although the in-memory part of the database might need to be reloaded. Or it might be more permanent (e.g., the file system has filled up), and more drastic action is required, although determining what to do could take a while. It's possible that there's a hardware failure (with 300 nodes and 600 disks, we experience regularly). Replacing the node will be required – but that may mean that the database has to be reloaded.

The tricky part here isn't the choice of which recovery action to invoke – with global knowledge, there are approaches to handling this problem (e.g., [Joshi2005]). The difficulty is that several Prato components may simultaneously determine that there is a fault – e.g., the node pool manager, the WX2 instance manager, and the WX2 node manager. Which of them should proceed? And how is this choice to be represented? We consider this a relatively simple instance of policy (de)composition, and yet it doesn't seem to be easy to solve without resorting to centralized decision-making. Policy composition seems to be an open question. We believe that to receive all the benefits of SOA, policies must compose.

In Prato, we have the luxury of resorting to a central controller if we must. But this is not always possible: what if the components were independent services provided by distinct entities, with separate administration, management, and fiscal domains. For example, we are planning on replacing Prato's NodePoolManager with a stub that delegates most of its work to an external physical resource-set service. Failure modes in such complicated systems are essentially an emergent property of the system. How do we develop policies for dealing with failure modes that we cannot predict will emerge?

4. Current status

We have completed the Prato service architecture design, have implemented a subset of the design, and are implementing the remainder of the prototype. Prato is already capable of allocating a database on demand (albeit with some manual assistance), and it already has some internal customers.

We are currently working with the internal Prato customers to validate our use cases, design, and implementation. Our initial focus is on customers with a relatively static corpus of data that they expect to analyze repeatedly, because this matches the strengths of the WX2 engine, which is optimized for business intelligence/analytics queries. We have some initial customers within HP Labs that fit this profile: one that periodically analyzes a ~1 TB corpus of customer data (provided by a major international retailer), another that periodically analyzes many months of Nielson click-stream data (~420 GB), and another group that are attempting the Netflix Prize challenge [Netflix2006].

Our initial production Prato service makes over 200 DL360 nodes and 2 TB of storage available for such customers in the form of one WX2 virtual database appliance per customer. The Prato service can also make multiple smaller appliances available, each one for a distinct customer. Each such appliance is provisioned on a disjoint node set, provisioned for its use from the cluster.

5. Discussion

Our initial goal is to make Prato be a robust, supportable, self-managing, lights-out service for the HP Labs community. Doing that will require constructing solutions to the research questions we have identified here. We're pretty sure that we have answers; what remains to be seen is how well those answers do in practice.

We believe that there is much research to be done in the failure tolerance area, including finding ways to do failure detection, diagnosis, and recovery. We are extending our work into the runtime task of failure detection, isolation, and recovery. All are significantly complicated when the control system is executing on the same distributed system that is being monitored. We have done some initial research in this area towards "automated finger-pointing" for distributed systems [Pertet2007].

Initial experience with our customers at HP Labs has made us realize that we may need to deploy analytics packages on the cluster that hosts the Prato virtual DBMS appliance, rather than elsewhere. Although Prato offers a direct ODBC feed from the database engine to its customers, rather than imposing the overheads of a web services interface, there are times when the bandwidth offered to the outside world from the cluster simply isn't high enough. Since we don't want to distort the appliance notion by running general purpose software on the nodes allocated to the database, we are experimenting with giving researchers access to other nodes in the cluster on which to run their analytics scripts. (This is another motivator for wanting Prato to migrate to being a customer of a nodes-on-demand physical resource provider service, which can be used to manage the allocation of machines in the resource pool to different needs.) We observe that the interfaces for such service composition have always been part of the Prato design, but the requirement to co-locate such services has arisen only after real-life customer experiences at HP Labs.

One of the attractions of Prato as a service is that it makes it possible to consider an ecosystem of services that can be constructed around it. For example, we see value in making data extraction and transformation be separable services that can be tied into Prato. We also see value in layering business analytics packages (e.g., like those offered by Business Objects, Greenplum, and Information Builders) on top of Prato. By making such functions into a SOA-based service, we believe it will be possible to reuse them and compose them in interesting ways. However, correctly composing the policies of such services stands as an open question.

6. Summary

Prato is a service provider that delivers database-management systems on demand to clients. Prato's benefits are two-fold: it provides a useful service to its clients, who are given rapid, flexible access to more capabilities than they could justify on their own behalf. It also acts as a vehicle for doing research on automated control and management of failure-tolerant distributed systems, and for learning how to solve the problems that will have to be tackled before on-demand provisioning of information services can become commonplace. The guiding principal of self-management in Prato is to maximize profit. This guiding principal informs the policies used to automate database instance design, price/admit customer contracts, and to plan recovery strategies from failures. Further research is required to understand how to incorporate risk into these policies, and how to compose policies of disparate services.

References

[Beckett2004] *HP Labs goes Hollywood*. Jamie Beckett. HP Laboratories, April 2004. http://www.hpl.hp.com/news/2004/apr-jun/nab.html, accessed April 2007.

[Enigmatec2007] *Execution Management System (EMS)*. Enigmatec Corporation Ltd. http://www.enigmatec.net/Execution_Management_System/, accessed April 2007.

[Joshi2005] K.R. Joshi, W.H. Sanders, M.A. Hiltunen and R.D. Schlichting. *Automatic Model-Driven Recovery in Distributed Systems.* In *Proc. 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pp 25-38. October, 2005.

[Keeton2004] *Designing for disasters*. Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase and John Wilkes. *Proc. of File and Storage Technologies (FAST'04)* San Francisco, CA, March-April 2004.

[Kognitio2007] *Kognitio WX$_2$*. http://www.kognitio.com/services/products/wx2.php, accessed April 2007.

[NeoView2007] *Neoview Platform – HP*. http://h71028.www7.hp.com/enterprise/cache/414444-0-0-225-121.html, accessed April 2007.

[Netflix2006] Netflix Prize. http://www.netflixprize.com/, accessed April 2007.

[Pertet2007] Soila Pertet, Priya Narasimhan, John Wilkes, and Jay Wylie. *Prato: databases on demand*. To appear as a poster in *The 4th IEEE International Conference on Autonomic Computing (ICAC'07)*. Jacksonville, FL, June 11-15, 2007.