Automated Multi-Tier System Design for Service Availability

G. (John) Janakiraman, Jose Renato Santos, Yoshio Turner
Hewlett-Packard Laboratories
{john.janakiraman,joserenato.santos,yoshio.turner}@hp.com

***Abstract*— Creating a cost-effective large-scale multi-tier Enterprise service requires judicious selection and configuration of infrastructure elements and mechanisms. The minimum cost solution that satisfies business requirements for service availability and performance should be identified. Emerging self-managed computing utility environments demand an automated solution to this problem. This solution must be integrated with the utility controller, which typically virtualizes resources for services, thus hiding from them information about the characteristics of the underlying physical infrastructure. In this paper, we present AVED, our initial version of an engine that automatically designs a cost-effective service infrastructure which will meet the service's availability requirements. AVED explores a design space consisting of multiple combinations of hardware/software configurations presenting various tradeoffs among cost, availability, and performance. AVED can be used to generate a complete picture of the cost-availability tradeoff for the infrastructure design. We also describe how AVED can be integrated with utility computing environments to improve the automation of service lifecycles.**

I. Introduction

Major systems vendors such as HP, IBM, and Sun Microsystems are vigorously pursuing the grand vision of delivering computing as a utility to dramatically improve the efficiency and simplify the management of information technology resources [7][15][9][3]. The idea is that a user who wishes to deploy an Internet or Enterprise service could issue a request to a computing utility, which in response would automatically allocate and configure appropriate resources from pools of compute, storage, and networking resources to create a secure, virtualized computing environment that realizes the service. Even more ambitious is the notion that a computing utility would automatically manage the provisioned service throughout its lifetime by dynamically tuning the design and deployment of the service's computing infrastructure in response to changes in service workload, component failures, etc. The computing utility would continuously manage the service infrastructure to ensure that service availability and performance are at levels that are adequate for the user's business mission.

A key component of such a self-managing computing utility is an automated design engine which would design a service's computing infrastructure and dynamically re-design it whenever necessary throughout the service's lifetime. In contrast, manual design and re-design processes are likely to become increasingly problematic as the use of resource virtualization becomes pervasive within utility computing environments. To illustrate this point, consider the impact of virtualization in Hewlett Packard's Utility Data Center (UDC) [7], a commercial solution that exemplifies the current state of industry progress toward realizing the utility computing vision. With its current functionality, the UDC makes infrastructure management simpler and more flexible by allocating virtualized compute, storage, and network resources (and services such as DNS) to users on demand. The UDC can improve service availability by automatically replacing failed resources in a user's environment with virtually similar resources from the physical pool. However, service availability is not entirely in the user's control because the virtualization layer hides information about availability properties of the underlying physical resources (e.g., failure correlation of servers attached to a shared switch that can fail[1]). Moreover, the trend is toward increased use of infrastructure virtualization and thus more abstraction of physical resource attributes. Since these attribute values are available only to the utility's control software, an automated design engine that is integrated with the control software can be used to determine the right amount of virtual resources to be allocated to a service and the right set of physical resources to be used, given high level specifications of desired performance and availability.

In this paper, we present AVED, an initial, proof-of-concept prototype which automates the design of a highly available system infrastructure for a service through exploration of a design space of resource and configuration alternatives. AVED targets the automated design of services that have the common multi-tier structure (e.g., web tier plus application server tier plus database tier). We describe AVED in the context of the UDC. We can envision future versions of the UDC that would integrate AVED into the utility control software for improved self-management functionality.

The design space that must be explored automatically by AVED can be large and consist of multiple dimensions such as choice of hardware components, software configurations such as database checkpointing frequency, use of redundancy through active, standby, or cold sparing, redundancy in network paths, use of software rejuvenation techniques, etc. Each choice within each of these dimensions presents a different tradeoff among availability, performance, and cost of ownership. The problem is to find a solution from this multi-dimensional design space that provides the best cost-

[1]In fact, with the UDC network topology, two network devices would need to fail before the attached computing devices are disconnected.

benefit tradeoff to the user. This tradeoff can be modeled with a utility function of cost, performance, and availability. In a simple case, the problem can be reduced to finding a minimum cost solution that meets the user's availability and performance goals specified as simple thresholds. We take this simple approach for our initial version of AVED.

In current practice, human system designers explore the design space by drawing on their expertise and experience to *manually generate* system design alternatives. To *evaluate* the availability of the generated design alternatives, designers use an availability modeling tool [4][2][14], and databases of component failure rates and repair times. The modeling tool predicts service downtime and the cost of downtime based on a business mission, which expresses for example that downtime during weekends is less costly than downtime on weekdays. The predictions are predicated on the use of best practice IT management [13] and thus provide an upper limit on the availability that can be achieved. Thus these tools are most useful for revealing the cost-benefit tradeoffs of different infrastructure options. With our approach, AVED *automatically generates* designs and evaluates them using an availability modeling tool inside an execution loop that iterates to find a design that meets availability and performance requirements at minimum cost. AVED can be faster than human designers in generating optimal designs. In addition, AVED may improve solution quality by covering a wider range of design alternatives than is usually feasible with a manual approach.

The rest of this paper is organized as follows. Section II describes how self-managing systems (e.g., UDC) could integrate support for automatic design and management for service availability. Section III presents the architecture of AVED, and Section IV presents an example showing its potential use. Section V briefly summarizes related automated technologies for high availability. Finally, Section VI presents our conclusions and future work.

II. Factoring Availability in Self-Managing IT Infrastructures

IT infrastructures powering a computing utility must be self-managing. The functional scope of these self-managing IT infrastructures must include automation for service availability. Prevailing solutions for IT infrastructure self-management have minimal or no functionality to automate the delivery of high availability for services. Our work is focused on extending IT infrastructure self-management solutions with support for automated availability. To discuss these extensions in context, we first identify the key components of a typical self-managing system by examining HP's Utility Data Center (UDC) solution.

The UDC is a programmable computing infrastructure with a utility controller that automates the creation, monitoring and metering of multi-tier server farms. To host a service in the UDC, a user must *describe* the service to the UDC through a web portal, identifying required hardware resources, the manner in which they should be interconnected, and operating

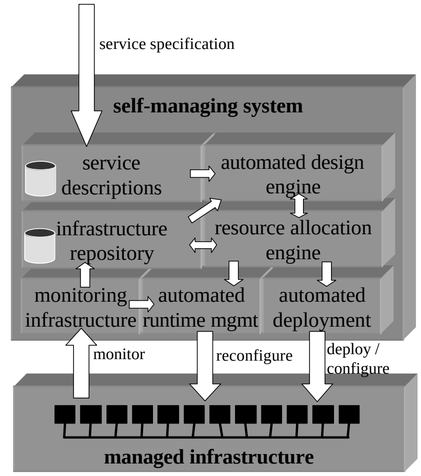

Fig. 1. Components of a self-managing system

system images. The utility controller maintains *infrastructure information* such as the hardware components that are part of the UDC's physical infrastructure and their physical interconnection topology. A *resource allocation engine* in the utility controller uses this information and the usage status of infrastructure resources to determine the set of physical resources that are free to be allocated to the service. Once the resources are allocated, the utility controller *automatically deploys* the farm by configuring network and storage components, and configuring and booting servers with specified disk images and operating systems. Furthermore, during service operation, the utility controller *monitors* all resources deployed in the farm and also performs *runtime management functions*. For example, if it detects any resource failure, it automatically deploys replacement resources. From this functional description of the UDC, we can derive that typical self-managing systems will be composed of these functional components: 1) *service description means*, 2) *infrastructure attributes repository*, 3) *resource allocation engine*, 4) *automated deployment*, 5) *monitoring infrastructure*, and 6) *automated runtime management*.

Next-generation self-managing systems will likely include richer implementations of these same functional components. In addition, since next-generation self-managing systems are likely to be driven by higher level service descriptions (e.g., specifying the service as a database with a minimum transaction throughput requirement, rather than specifying the use of a particular type of hardware resource), they will also need a *design engine* that maps these high level requirements to infrastructure requirements (e.g., determining the type of machine and the number of these machines based on the performance requirement). Fig. 1 illustrates these functional components of a self-managing system.

To automate for availability, it is necessary to extend the functional components described above. The extensions of components 1-6 above are fairly straightforward and are discussed further in [10]. In this paper we focus on the remaining component, *automated design engine*, which is responsible

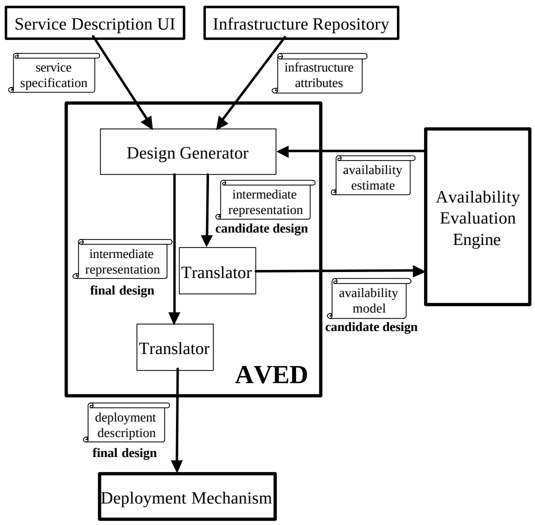

Fig. 2. AVED architecture

for selecting and configuring infrastructure components and availability mechanisms such that high level requirements are met. The work we describe in this paper represents our initial efforts in developing this automated design engine. The design engine must automatically explore several design alternatives and embed appropriate models of the environment to determine which design alternative will satisfy higher level requirements at minimal cost. The automated design engine will likely be implemented as a hierarchy of engines, each of which is responsible for the design/configuration (initial and subsequent re-design) of a subset of the overall environment. For example, a service that includes a database system may have an automated configuration/tuning engine associated with the database system in addition to a service-level design engine.

III. Automated Design for High Availability

AVED is our proof-of-concept initial prototype of a simplified automated design engine for utility computing. AVED identifies and describes the minimum cost design that satisfies the user's requirements. The key challenge for the architecture and operation of AVED is to devise workable techniques for modeling and searching the service design space, including the various combinations of availability mechanism options that can be used in each design. In this section we present our progress so far in meeting that ambitious challenge.

The overall architecture of AVED is shown in Fig. 2. The inputs consist of a *service description* and an *infrastructure repository*. They describe the design space by specifying the structure of the system design and the various availability mechanism options. AVED searches the design space in an execution loop, generating a series of designs that it feeds to an availability evaluation engine for analysis. The designs AVED generates are represented internally using a data structure that is independent of the evaluation engine. A translation module inside AVED converts this intermediate representation of a design into an availability model that is input to an availability evaluation engine such as AVANTO[2], Möbius [2], SHARPE [14], etc. By using this approach, AVED can be interfaced to a variety of availability modeling tools simply by customizing the translator to the evaluation engine. AVED translates the optimal design's intermediate representation to a format useable by an automated service deployment engine (see Fig. 2), which instantiates the service on actual hardware.

The service description input to AVED has two purposes. First, it specifies the high-level performability requirements that must be satisfied by the service. The performability requirements currently have a simple specification consisting of just two parameters: the minimum acceptable *performance* (in service-specific units such as transactions per second for the expected type of transaction), and the maximum annual *downtime* allowed. We use the term *annual downtime* or simply *downtime* to indicate the expected time a system will be unavailable in a year. We consider a system is unavailable whenever the number of active resources is not sufficient to achieve the service performance requirement. Second, the service description describes the service structure by listing the tiers that are to comprise the service implementation, the candidate resources that can be used in each tier, their performance characterization, and indication if the service could be deployed in a clustered configuration[3]. For example, the following attribute-value pairs describe an example three-tier structure:

```
tier_name=Web-Tier
  resource=ResourceA  cluster=True singleload=100  nmax=25
  resource=ResourceB  cluster=True singleload=300  nmax=25

tier_name = Application-Tier
  resource=ResourceC  cluster=True singleload=200  nmax=25
  resource=ResourceD  cluster=True singleload=600  nmax=25

tier_name = Database-Tier
  resource=ResourceE  cluster=False singleload=500  nmax=1
  resource=ResourceF  cluster=False singleload=1500 nmax=1
```

For each tier, the specification provides a list of candidate resource types that can be selected to design the tier. Each distinct resource type corresponds to a unique combination of hardware and software components (described in the next paragraph). In this example, the web tier can be implemented either with resources of the ResourceA type or with resources of the ResourceB type. Each design generated by AVED chooses exactly one of these options for each tier. For each candidate resource type, a boolean flag *cluster* indicates if the resources can be used in a cluster configuration. If not, active spares cannot be used. In this example, active spares cannot be used for the database tier, which is limited to a single active node, but they can be used for other tiers. In addition, the description of a resource includes a characterization of its performance under the service's workload. We currently use a simple linear saturating performance model for cluster configurations. The parameter *singleload* indicates the performance, in service-

[2]An HP tool used by HP Services to evaluate infrastructure availability

[3]In the future, we would additionally like to enable specification and modeling of failure dependences between application components.

specific *load units*, of a single resource, and the performance of a cluster of resources of the same type[4] scales linearly with the number of active resources until saturating at a cluster of size *nmax* active resources.

The infrastructure repository input to AVED describes the availability properties and costs of the various resource options. Each resource consists of a collection of hardware and software components. In our current model any component failure causes the resource that contains it to be down, and we intend to capture more general failure dependency relationships in future versions of AVED. The following example defines two resource types, each consisting of three component types (in this case: hardware, OS, and application):

```
resource=ResourceA   MachineB Linux WebServerX
resource=ResourceB   MachineA Unix  WebServerX
```

For each component type, the repository specifies its cost and availability properties. We next explain how these are described using the following example specification of a hardware component type:

```
component=MachineA cost_cold=1000 cost_active=1100
  perm_mtbf=650d failover_used=true failover_duration=2m
     repair_mttr=15h repair_cost=580
     repair_mttr=6h  repair_cost=1500
  tran_mtbf=75d failover_used=false
     repair_mttr=30s  repair_cost=0
```

Cost: The annualized cost of a component is given for its various operating modes. In this example, the component can be either powered off if it is a cold spare ($cost\_cold = 1000$) or powered on if it is an active spare ($cost\_active = 1100$). The cost difference may account for the electrical power costs that are incurred only when the hardware is powered on. As another example of mode dependent cost, an application software component might incur cost only when it holds a floating license. The annual cost of a component is the sum of the annual cost to operate it in its operating mode and the initial (capital) cost of the component annualized by dividing by its useful lifetime in years. The annual cost of a resource is the sum of the costs of its components in their corresponding operating modes.

Failure modes: A component can have multiple failure modes. This example indicates a permanent failure mode with MTBF $perm\_mtbf = 650$ days and a transient failure mode with MTBF $tran\_mtbf = 75$ days.

For each failure mode, its failover behavior and repair options are specified as follows:

Failover: For each failure mode, failover to an available cold spare is enabled if the mode's *failover_used* flag is true, in which case failover requires time *failover_duration* to complete. In this example, a permanent failure triggers failover to a cold spare resource in two minutes, but a transient failure does not trigger failover.

Repair options: Multiple repair options can be specified for each failure mode. A design selects exactly one of these repair options for each component instance, and the selection applies to all peer components in the same tier. Each repair option is described by the Mean Time To Repair (MTTR) it enables[5], and the annualized cost per node of choosing the repair option. Availability mechanisms with continuous parameters, such as checkpointing with configurable checkpoint frequency, can be represented as multiple discrete repair options. In the example above, permanent failure has two repair options. One costs \$580.00 per node and completes repair with $mttr = 15$ hours, and the other costs \$1500.00 per node but completes repair in only 6 hours. These could represent, for example, maintenance contracts that differ in cost and in the response time of the hardware repair staff. For transient failure, there is only one option, which corresponds to resetting the hardware. It takes 30 seconds for the hardware to come back up, and there is no cost to have this option.

For each repair option, a performance degradation parameter (not shown in this example) can optionally be specified. This parameter indicates the degradation (as a percentage value) of the component's performance during fault-free operation as a result of having the repair option in the design (e.g., checkpoint overhead). Although this parameter is part of the infrastructure repository, the precise value of performance degradation is likely service-specific, and therefore this value should be monitored and verified during normal operation of a deployed service or prior to deployment through offline evaluation. Performance degradation also may occur after a component is repaired but before it has been completely reintegrated into the service infrastructure. This degradation may affect one or more components in the cluster during this integration process. In the future we plan to specify and model this type of performance degradation during recovery.

Preemptive maintenance: In addition to failover behavior and repair options, we see the need to describe the impact of preemptive maintenance on availability. For example, the use of software rejuvenation [8] can have the effect of improving MTBF for a failure mode, as opposed to repair options which have impact on MTTR, not MTBF. In the future we plan to define parameters for describing preventive maintenance options.

The number of components of each type that can be used to build a design may be limited, particularly in a utility computing environment in which existing hardware is intended to be used instead of purchasing and installing new hardware. For now, an optional parameter (not shown in this example) is used to indicate the maximum number of components of each type that can be selected for a design. In the future we intend to add the ability to describe more sophisticated resource constraints.

To generate a design, AVED makes a series of design choices as it incrementally builds an intermediate representation for the generated design. This representation has a

[4]For simplicity, we currently limit each tier to be homogeneous in resource type.

[5]Currently, we include failure detection latency into the MTTR values. We intend to extend AVED to enable exploration of alternative detection mechanism options that differ in detection latency, coverage, accuracy, and performance degradation impact.

hierarchical structure of tiers, resources, and components. AVED selects exactly one resource type for each tier and the number of active resources of that type to instantiate. Some of these active resources may be active spares, not needed to meet performance requirements unless some resource fails. In addition to choosing the active resources, AVED selects the number of cold spare resources for each tier[6]. AVED selects exactly one repair option for each failure mode of each component of each resource, ensuring that the selection is identical for resources that are peers in a tier. As AVED creates the hierarchical intermediate representation of a service design, it also calculates the design's cost, which is the sum of the cost of the components and the cost of the repair options selected for the components. The cost and availability of all the designs determine which one is optimal.

AVED's design space search strategy is designed to minimize the number of designs that need availability evaluation, since this is the most time consuming operation. The search strategy is composed of two stages. In the first stage each tier is evaluated independently to compute its optimal design. In the second stage the solutions for the tiers are combined to determine the global optimal design [10].

IV. Example

We illustrate the value of AVED using a simple example scenario: designing the Application Tier of an Internet Service for high availability.

In this scenario, the following design dimensions are explored by AVED: **1)** resource type, **2)** number of extra machines, **3)** state of extra machines **4)** selection of maintenance contract. We assume the infrastructure supports two different types of machines: a dual processor machine (machine type M-A) which can run Linux and a more powerful 8-way machine (machine type M-B) which can run a proprietary version of UNIX. In addition, we assume we have a choice of two different types of J2EE Application Server software, AS-A and AS-B, that can be installed on either hardware platform. By combining the two hardware options with the two software options, AVED can explore four different resource options. Extra (redundant) machines added to improve availability could be used in two states: **1) active:** the machines are added to the cluster and are always operational, except when they fail. **2) cold spare:** the machines are turned off and only turned on to replace a failed machine. Cold spares have lower cost than active resources because both operational and software licence costs are avoided. However, service downtime is incurred during failover to a cold spare. Finally, the repair options for failed hardware consist of four different maintenance

[6]Currently, designs generated by AVED cannot include both active spares and cold spares in a single tier. We plan to remove this restriction in the future.

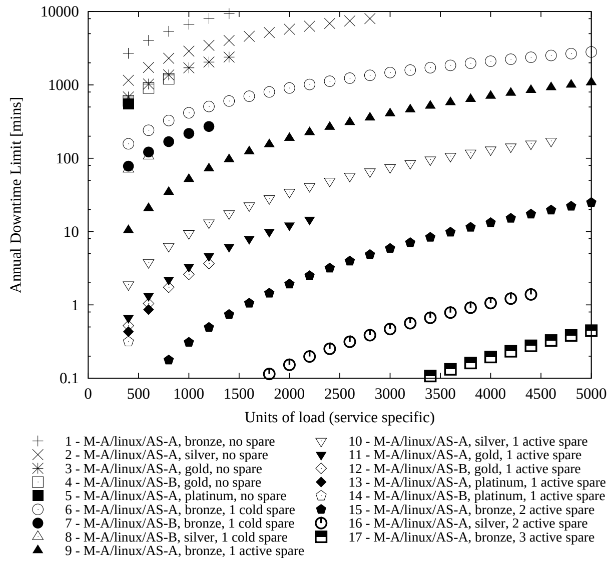

Fig. 3. Optimal solution for a range of service requirements: load and annual downtime limit.

contracts[7] that can be purchased from a service provider. Each contract has a different cost and provides a different response time for on-site technical support necessary to repair hardware failures. We made an effort to choose realistic input parameters for this example. We obtained failure rates or MTBF values for hardware components from the manufacturer historical database. We selected costs and response times for service maintenance contracts offered by the hardware vendors. Software and hardware costs were obtained from vendors' published prices. However, software failures rates were estimated based on the authors' own intuition, since this data was not readily available.

We have used AVED to identify the optimal designs for this example scenario over a range of service performance and availability requirements. Fig. 3 shows these optimal designs as a function of the performance requirement (Units of load) and the availability requirement (Maximum annual downtime[8]). The designs are represented in Fig. 3 by a tuple *(resource,contract,redundancy)*, where *resource* indicates the selected type of resource, *contract* indicates the selected Service Maintenance Contract, and *redundancy* indicates the number of spares and their state (active or cold spare)[9]. To facilitate the discussion in the rest of the paper we also refer to the designs using numbers as identified in Fig. 3. The load

[7]Maintenance contracts are more meaningful in a manually designed and managed IT environment than in a self-managed IT environment. We have chosen to show this in our example because we have not yet completed obtaining realistic data for other mechanisms such as dynamic resource replacement, database checkpointing tuning, and software rejuvenation which are applicable in a self-managed environment.

[8]We refer to maximum annual downtime simply as downtime in the rest our discussion.

[9]Note that the same design can use a different number of machines depending on the load, i.e., design with *m* spares has a fixed number of redundant machines, in addition to the number of primary machines which can vary as function of a load.

range shown on the *x axis* varies from 400 to 5000 *units*[10], which corresponds to a range of 2 to 25 nodes. The *y axis* shows the range of practical annual downtime values, from a fraction of a minute to 10,000 minutes, i.e., approximately one week[11]. In the two-dimensional space of requirements mapped by the performance requirement and the availability requirement, each curve corresponds to a particular design that is cost optimal for all requirement points above the curve (and points on the curve) and beneath the immediately higher curve. Furthermore, the curve plots the downtime estimate for this design at various load levels where it is the optimal solution. Therefore, for requirement points above the curve, the downtime estimated with this design is less (i.e., better) than the requirement. For example, for a requirement ($load = 1000$, $downtime = 100$) in Fig. 3, the curve immediately below this point corresponds to the optimal design (number 9), which has downtime of approximately 50 minutes.

The results in Fig. 3 show that despite the small size of our example design space, the number of optimal solutions distributed across the requirements space is large and would be tedious to evaluate manually. The results also show that AVED filters out suboptimal solutions. For example, design 3 (M-A/linux/AS-A, gold, no spare) is not selected for loads above 1400 *units*. For loads above that, design 6 (MA-A/linux/AS-A, bronze, 1 cold spare), which provides lower downtime, is selected instead of design 3. For low loads the extra cost of the *gold* maintenance contract is lower than the cost of an additional resource and design 3 is the preferred design when its downtime satisfies the service requirement. As the load increases, the extra cost of the *gold* contract becomes higher than an extra resource, since the cost of a maintenance contract is proportional to the total number of machines it covers. Thus for higher loads it is better to use an extra node than to pay for a higher maintenance contract. In fact, we observe in Fig. 3 that as the load increases, all the selected designs use the lowest cost *bronze* maintenance service contract.

As shown in Fig. 3, the more powerful machine M-B is never selected. This is expected since we assumed linear scalability for the application, and the low end machines have a better cost-performance ratio (i.e., lower cost per unit of load). However, the situation may be different if the application scales sublinearly with the number of nodes. In such cases, beyond a certain load, it may be possible to achieve a better cost-performance ratio by using a lower number of more powerful machines than a higher number of low-end machines.

We observe in Fig. 3 that the downtime estimated for a particular design increases with load. This is expected since higher load levels require a larger number of nodes which results in a higher failure rate (because if any of these nodes fail, the service cannot meet its minimum performance requirement and the service is considered down). Thus in self-managed environments such as the UDC, where the infrastructure can be dynamically changed to adapt to load fluctuations, the optimal design may change as the load changes. For example, consider a service that tolerates at most 200 minutes downtime and has an initial expected load of *400 units*. From Fig. 3, design 6 is the optimal design. However, if the loads increases to *1200 units*, the optimal design changes to design 9.

Although the curves shown in Fig. 3 enable the selection of the optimal design for a given performance and availability requirement, the knowledge of the cost associated with each design can help to make a better design choice. AVED can be used to generate plots of the costs of optimal designs at various levels of availability and performance requirements. Such plots reveal the tradeoffs among cost, availability, and performance that must be understood to make a judicious design choice. For example, in some cases a large improvement in downtime can be achieved with a low additional cost. Alternatively, slightly relaxing the downtime requirement can significantly reduce the cost overhead for availability citeavedtech.

In an automated dynamic environment, the tradeoff between cost and availability must be evaluated automatically. For these environments, we envision the use of a utility function, specified by the service, which estimates the cost associated with each value of annual downtime (e.g., the expected loss in revenues when the service is not operational). Given this utility function, a system may automatically select the best design to minimize the total cost (the sum of the cost overhead for availability and the cost of lost revenue from downtime).

V. Related Work

The idea of automating the design and configuration of systems to meet user's availability requirements is relatively recent. We are only aware of a few examples, each of which is focused on a limited domain. The Oracle database implements a function that automatically determines when to flush data and logs to persistent storage such that the recovery time after a failure is likely to meet a user-specified bound [12]. Researchers at HP Labs have proposed automated design of storage systems to meet user requirements for data dependability, which encompasses both data availability and data loss [11]. Our research addresses the automated design of multi-tier systems which include databases, storage systems in addition to other hardware/software components. Hence, technologies for automating subsystems such as databases and storage systems will be integrated as elements of our overall solution for automating availability.

Most other work on system automation for managing availability has been limited to automated monitoring and automated response to failure events and other such triggers. For example, cluster failover products such as HP MC/Serviceguard [5] SunCluster [16] and Trucluster [6] detect nodes that fail, automatically failover application components to surviving nodes, and reintegrate failed nodes into active service when they recover from failure. IBM Director [1]

[10]Our unit of load is associated with an arbitrary unit of work per unit of time that is meaningful for the specific service, as for example transactions/sec.

[11]We believe it is not useful to explore very low downtime values, since AVED only models infrastructure availability. When the infrastructure availability is reduced to very low levels, other external factors, difficult to model and characterize such as human error, environment effects, etc., will dominate.

detects resource exhaustion in its software components and automates the rejuvenation of these components at appropriate intervals. Various utility computing efforts underway will also automatically detect failed components and automatically replace them with equivalent components from a free pool [7][15][9].

VI. Summary and Future Work

Management for service availability must be a critical component in self-managing IT infrastructures, since degraded service availability can lead to significant loss of business. Our research aims to advance the state of the art in self-managed systems (an area largely in its infancy) by integrating support for automatically managing service availability. We believe automation for availability is most valuable and usable if driven by high level availability requirements that are intuitive at the service level. To enable goal-driven automation, our approach proposes several extensions to emerging self-managing system designs citeavedtech. In particular, the self-management system must include a design function that automatically generates design alternatives, builds their availability models and evaluates them to identify the best design that meets specified requirements.

Our current efforts in this direction have been focused on the automated design engine. We have described the architecture of our initial version of the automated design engine, AVED, including a model for describing the availability characteristics of a service and the infrastructure. AVED maintains system designs internally in an intermediate form of representation, and these intermediate representations are then translated and input to an availability evaluation engine. We have examined a simple example scenario using AVED to illustrate the usefulness of AVED. The example exposes varying tradeoffs between the availability knobs across the design space and requirement space, justifying the value of automated design space exploration. In some cases, designs with small differences in their cost may have substantial differences in their availability. The automated design engine can take advantage of such tradeoffs if services specify their requirements using utility functions (e.g., one that describes the business cost of various downtimes) rather than as particular points (e.g., 5 min annual downtime maximum). In addition, parameters such as the number of resources running the service and repair times, which will change dynamically (the number of resources running the service will likely change with dynamic changes in the applied load in a self-managed environment), are seen influencing the selection of the design. This indicates that the self-management system must automatically reevaluate and reconfigure designs in response to changes in such parameters.

We intend to enhance AVED in several ways. To address overall service availability, the design engine must examine the impact of the network and storage subsystems. We will be extending AVED to factor network topologies (LAN), application placement in the network and network failures and recovery. We also plan to integrate AVED with an automatic process for storage system design and management for data dependability[11]. We also intend to make AVED's design space richer by adding consideration of several other knobs, including configuration parameters of the database engine (e.g., its checkpointing frequency), configuration parameters of the application server (e.g., the location where persistent state is replicated), the use of virtual machines to host multiple application components on a single hardware platform, and software rejuvenation. We are also looking to make small changes such as relaxing the restriction in our current implementation that each tier be homogeneous, and permit tiers with heterogeneous components.

References

[1] Castelli, V., Harper, R. E., Heidelberger, P., Hunter, S. W., Trivedi, K. S., Vaidyanathan, K., and Zeggert, W. P. Proactive management of software aging. *IBM Journal of Research and Development 45*, 2 (March 2001), 311–332.

[2] Clark, G., Courtney, T., Daly, D., Deavours, D., Derisavi, S., Doyle, J. M., Sanders, W. H., and Webster, P. The Möbius modeling tool. In *9th Int'l Workshop on Petri Nets and Performance Models* (Sep 2001), pp. 241–250.

[3] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. Grid Services for distributed system integration. *Computer 35*, 6 (2002).

[4] Hewlett Packard Company. *Availability advantage*. (http://h18005.www1.hp.com/services/advan-tage/aa_avanto.html), January 2003.

[5] Hewlett Packard Company. *HP MC/ServiceGuard*. (http://www.hp.com/products1/unix/highavailabil-ity/-ar/mcserviceguard/index.html), January 2003.

[6] Hewlett Packard Company. *TruCluster software*. (http://www.tru64unix.compaq.com/cluster/), January 2003.

[7] Hewlett Packard Company. *Utility computing*. (http://devresource.hp.com/topics/utility_comp.html), January 2003.

[8] Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. Software rejuvenation: analysis, module and applications. In *25th Symposium on Fault Tolerant Computer Systems* (Pasadena, CA, June 1995), pp. 381–390.

[9] International Business Machines, Inc. *Autonomic computing*. (http://www.ibm.com/autono-mic/index.shtml), January 2003.

[10] Janakiraman, G., Santos, J. R., and Turner, Y. Automated multi-tier system design for service availability. Tech. rep., Hewlett Packard Laboratories, May 2003.

[11] Keeton, K., and Wilkes, J. Automating data dependability. In *10th ACM-SIGOPS European Workshop* (Sep 2002).

[12] Lahiri, T., Ganesh, A., Weiss, R., and Joshi, A. Fast-Start: quick fault recovery in Oracle. In *ACM SIGMOD* (2001), pp. 593–598.

[13] Office of Government Commerce. *ITIL Service Support*. IT Infrastructure Library. The Stationery Office, United Kingdom, June 2000.

[14] Sahner, R. A., and Trivedi, K. S. Reliability modeling using SHARPE. *IEEE Transactions on Reliability R-36*, 2 (June 1987), 186–193.

[15] Sun Microsystems, Inc. *N1: Revolutionary IT architecture for business*. (http://www.sun.com/soft-ware/solutions/n1/index.html), January 2003.

[16] Sun Microsystems, Inc. *Sun[tm] Cluster*. (http://www.sun.com/software/cluster/), January 2003.