Utilification

John Wilkes, Jeffrey Mogul, and Jaap Suermondt

HP Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, USA
john.wilkes@hp.com, jeff.mogul@hp.com, jaap.suermondt@hp.com

Utility computing has the potential to revolutionize the way we purchase, organize, and distribute computational power and services. It will do so by offloading resource provisioning to centralized sites that can benefit from economies of scale, careful, failure-resilient construction, flexibility and changeability of hardware choices, and scalable and business-driven management techniques. But that promise is useless unless we can move applications from traditional computing environments into utility ones, where the applications are fronted by service interfaces and resource flexing is the norm. This paper argues that such transformations are worthy of study and effort, and suggests that the systems community has a great deal to offer to them.

Utility computing

Utility computing systems allow rapid responses to changing computing demands, new business processes, disasters, and other events. They are quite the craze these days. HP has its Adaptive Enterprise strategy, which encompasses business-process agility and its Utility Data Center (UDC); Sun Microsystems has its N1 system ("managing *n* computers as 1" [14]); and IBM is offering on-demand computing. Many startup companies are working in this space, too.

All are pursuing much the same ideas: it is better to harness computing to the service of business than the reverse; it is better to streamline the workload of scarce, highly-paid system managers; and it is becoming essential to respond rapidly to changes in demand and situation in today's commercial and institutional climates. The relentless commoditization of hardware has led to more appreciation of the importance of tools for managing systems for more direct business goals – not just increased hardware utilization.

The terminology in this rapidly-developing field is still inconsistent. We will simply define utility computing as the reliable, resilient, scalable provision of computer-based services, as and when needed, in response to internal and external events. This goes beyond the adaptive provisioning of computer hardware, whether virtualized or not, to include application-level services.

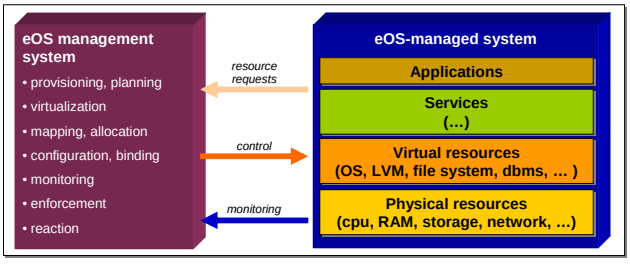

Figure 1: the eOS architecture.

The eOS architecture [16] is typical of the kind of structure that results: a layered stack of services plus a management infrastructure "on the side". A more sophisticated example is HP's *Darwin reference architecture* [7], which takes things one stage further by including business processes in the control loop.

We do not intend to argue the merits of different approaches to self-managing, utility computing systems: they are here to stay, and their benefits are compelling. Instead, we wish to argue the importance of the too-often neglected processes by which computing technology is adopted and made useful in the real world.

Utilification

The goal of this paper is to raise awareness of one such process: how existing applications are brought into a utility computing environment and helped to benefit from it. We call the process *utilification*[1].

Once an application is utilified, it is expected to flex its abilities dynamically as demands change; to put up with being assigned to different resource instances (processors, networks, storage), and to participate in the application- and business-level measurement and feedback-control system.

We used to think that utilification was simple. You seek out the application of interest, shut it down; and bring it up in the new environment. You might need to wrap some control-loop stuff around it, to measure throughputs and response times and map these onto the resulting resource needs, but the process was basically straightforward, even if effort-intensive.

And then we watched the actual process of utilifying an application – the one used by DreamWorks to render the animation for Shrek 2 on an HP utility computing environment [9]. The HP system has several hundred

[1] Thanks to Patrick Goldsack for coining the word.

Linux nodes and a few terabytes of data storage. It runs one large parallel batch application – image rendering.

What surprised us was that this application needed a local LDAP server for clients to access their NFS auto-mount tables. It never occurred to us that such a service would be needed in a UNIX environment. This insight led to other questions: How many other items like that LDAP server were there to think about? How many of them were caught on the first attempt? And, how much harder would this be in more complicated environments, where a single application was being teased out from a morass of existing ones? Consider the case of utilifying a complex business process like supply chain management running on SAP, on top of Oracle, across a failure-tolerant cluster of high-end multiprocessor machines: just how hard would this be?

Mutual migration

At one level, part of the process is like porting an application to a new architecture – but now the architecture is at a higher level than an instruction set: it's at the level of application component and service building-blocks; of network access and discovery services; of control-system APIs and larger business systems.

Further thought suggests that the process is a two-way one: the application can be brought towards the utility computing infrastructure, but the infrastructure can also be pushed towards the application (Figure 2).

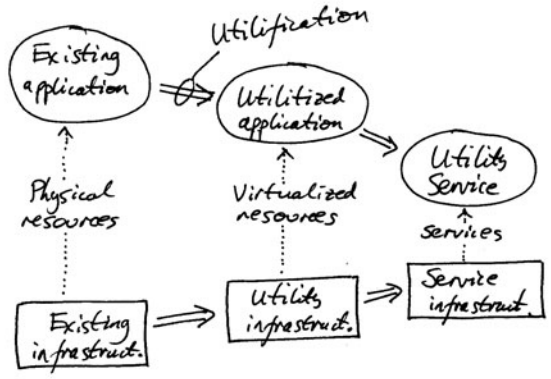

Figure 2: evolving applications and environments towards each other as part of utilification.

HP's UDC [8], for example, has taken the first step down this path by providing virtual data centers, each isolated from its peers by security barriers such as network VLANs and careful control of the storage area network. Similar approaches are employed in data center or application *consolidation* – the process of bringing applications running on multiple, dispersed servers together to run on a shared central system. (A common technique is to use "physical partitions" of the larger server.) This simple, isolated-environment approach to consolidation works quite well for complete application suites that were already well-isolated from their initial context, but it does little to address other kinds of setups.

The following sections examine some of the steps of bringing an application into the utility context. Each has technical challenges that represent opportunities for further research with direct, practical applicability.

Blueprinting and assessment

The first step in application bring-over is discovery: what application components are there? How are they configured? What are their dependencies on each other, and their environment?

This process is sometimes called *blueprinting*. There are already some blueprinting products on the market, using the technique to understand and report on an environment, support data center consolidation, or improve configuration change management. For example, Collation's Confignia "reveals the run-time configurations and interdependencies across application components, system services and network elements" across multiple tiers [5], and Cendura's Cohesion provides similar features, and specifically addresses change detection and auditing [4].

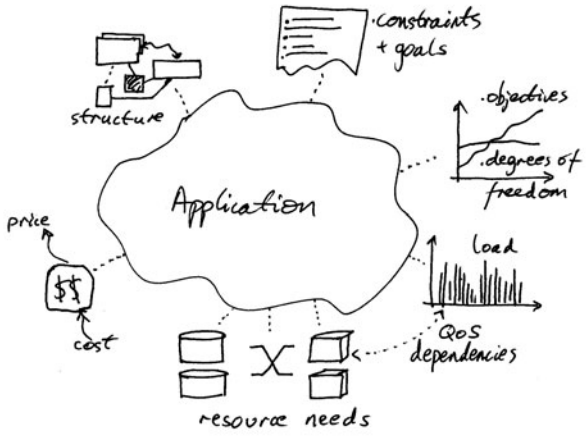

Figure 3: the assessment process.

The discovery process starts with learning about the components, and how they are connected – but it needs to go further. Figure 3 diagrams some of these additional elements; we call this extended form of blueprinting *assessment*. It includes gathering data on:

- how the application's behavior changes in response to varying workloads, changing resource availability, security and denial-of-service attacks, and hardware and software faults;

- how the application's resource needs change as a function of scale and usage across a range of external loads or other events (e.g., an internal security scan) ;
- whether and how the application can extend and shrink ("flex") itself to exploit additional resources;
- how much of the application's surrounding environment needs to be replicated for it to thrive; and
- information about any constraints that need to be imposed on the application, or that it imposes on its surroundings.

QoS-based sizing

One of the (many) hardest parts about assessment is the construction of a mapping from offered load to application resource needs. The goal here is to automate the process of deciding how to achieve a given application-level Quality of Service, or QoS. (See [17] for a storage perspective on this, and [2] for a recent distributed-system one.) A particularly important Quality is performance – typically measured by throughput or response time.

This is tricky enough to do with a single, monolithic application that has no external points of control. Even simple applications may need black-box techniques (e.g., [1], [3]) – it gets much harder when the application is composed of a set of components, each of which has multiple tuning parameters, as all real-world ones seem to be.

A related open question is how to set the "QoS budget" for each separately-tunable component: should a 100ms overall response-time budget be split 20:80 between two components, or vice versa? What if the resource demands of these two alternatives lead to very different costs? What if the cheapest solution is also the most susceptible to external disturbances or mis-estimations of the load?

Resiliency

This leads to a related thought: the utilification process is an ideal point at which to ask questions such as "how much application-level resiliency is needed?" and "is now the time to increase it?" Resiliency means different things to different people – and different applications – but it generally encompasses notions of availability (percentage uptime), performability (probability of achieving a given performance level), and reliability (resistance to data loss or corruption).

Redundancy and replication are the single commonest approach to achieving increased resiliency. These techniques can be applied at many levels: from storage systems (e.g., [12]), to application components (which will alter the relationships between the pieces), to complete application instances. Alternatively, instead of using redundancy and replication to avoid failures, it may be more fruitful to develop better techniques of detecting and recovering from them [5]. Making these design decisions is non-trivial; to make matters worse, the results will probably impact the results of the QoS-based sizing step, so that may have to be revisited.

Flexing

Some applications naturally lend themselves to resource flexing: adding additional servers to a loosely-coupled "embarrassingly parallel" scientific application, for example. But others are trickier.

An application component that cannot be replicated for performance, such as a database that cannot be partitioned, can perhaps be migrated to a faster computation or storage node. Of course, this brings its own difficulties: how to migrate an application while it is up and running, for example. Most application-migration systems of which we are aware assume implicitly that the target resource set is merely a clone of the original, and leave open the question of how to adjust the application control parameters simultaneously to reconfigure it for its new environment.

Flexing may also mean shrinking – either because demand for the service has fallen off or because another service has a more compelling need for a resource. Shrinking is much harder, and generally is something we prefer to stick our head in the sand about; yet learning how to do it is an integral part of utilification.

Dynamic control

One of the goals of utility computing is to enable "application agility": rapid adaptation to changing circumstances. This is typically pursued in the context of a target QoS level, or Service Level Objective (SLO), typically as part of a Service Level Agreement (SLA).

Open questions here involve how to best specify, capture, and review a target QoS level – bearing in mind that the specification may need to have enough information in it to allow machine-mediated resource allocation decisions without human involvement. Recently, there has been a new focus on business value in QoS metrics and SLAs: what matters is the business value of higher throughput and the cost of downtime, not just achieving arbitrarily-specified performance levels. Ultimately, the goal is to operate at a point of indifference between better QoS (performance, availability, etc) and higher cost [12].

The utility control infrastructure may need to allocate or withdraw resources from the applications it supports in

order to achieve application performance QoS goals, and so it needs an appropriate control loop to decide when to do so. We have already mentioned the modeling aspect of utilification (what to do to achieve a new QoS level), but that still leaves the monitoring aspect (What is the offered load? What are the achieved responses?), and decisions about when to apply changes, and (perhaps) in what order. All of these need grafting into existing applications, or a way to approximate the desired degree of finesse solely by means of externally-visible metrics.

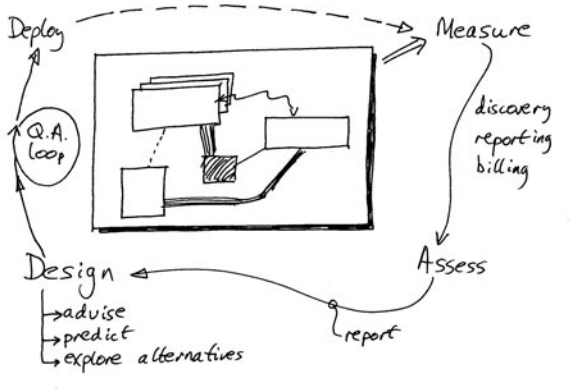

Figure 4: a canonical control-loop structure.

A simple control loop typically contains elements akin to those in the figure here. Nested control loops make this both more challenging and offer new opportunities: they can be used to provide fine-grained control to augment the higher-level goals (e.g., by throttling I/O rates during data migration). But they may also hide some of the effects of lower-level responses: consider a system that manages to sustain a target response time across a wide range of offered loads, but fails catastrophically as soon as those limits are exceeded.

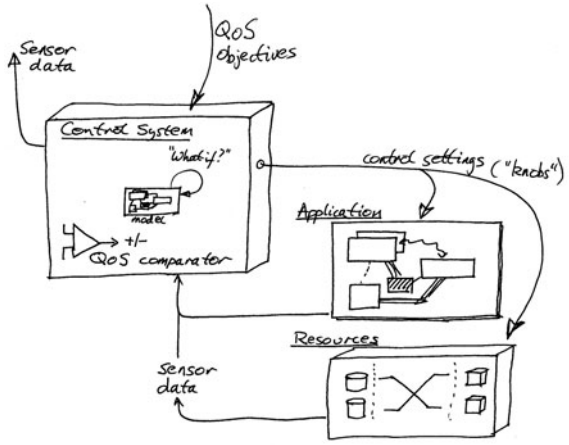

Figure 5: control system internal structure.

The control-loop structure is a well-established one. Its relevance to the utilification problem is two-fold:

1. it relies on a model of the target system under control, which has to be obtained or inferred somehow [11];
2. it may provoke unexpected interactions between control loops at different levels, and the likelihood of this should (ideally) be probed for during the utilification process.

For example, database systems like Oracle 10g [14] are taking on more of the characteristics of a complete computing environment, capable of reallocating resources to changing needs. If this is layered on top of a dynamic resource provision infrastructure, which level of the system is to make flexing decisions?

Trust and security

Utility computing enables the possibly of sharing resources across mutually-distrusting customers. Indeed, it may require this if it is to achieve its full economic benefits. Such sharing clearly requires both performance and security isolation between the customers. Appropriate security techniques are well known; what turns out to be hard is building customer trust. Customers of a utility need to agree that the right mechanisms have been deployed, that no loopholes exist, and that the mechanisms will achieve the desired results. This is a trust issue, not just a security one.

Since even the best-designed, best-run systems can have security holes, the ultimate recourse remains contractual agreements to compensate customers for the consequences of loss or damage. In turn, the utility provider's lawyers or insurers need to trust that appropriate risk mitigation techniques have been correctly deployed before they will sign such a contract.

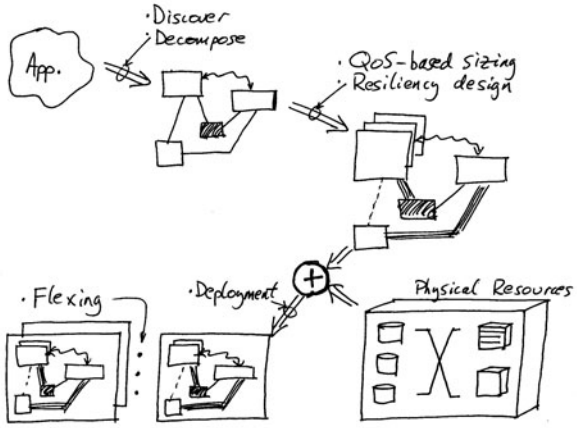

Figure 6: the complete utilification process.

Deployment

We can now pull this whole process together, as shown in Figure 6.

Of course, this quick survey elides a great deal of complexity in the individual steps, as well as a larger process: that of making appropriate business and customer-facing design choices. Both offer challenging opportunities for improving the tie between IT infrastructures, the way they are used, and the end goals of an organization.

Service-level interfaces

After an application has been utilified, one possible next step is to extend its functionality to provide a service – that is, give the application an interface by which "external" customers can submit work to it.[2]

This moves us from an infrastructure that is focused on providing (virtualized) physical resources to a service-based one; the domain of dialogue is service-component relationships, not resource demand-supply. Other opportunities include ways to package up access portals to a service, charge for (or at least meter) their use, and providing security and performance isolation between competing service users.

Conclusion

Bringing applications into a utility computing world poses several hard challenges. Addressing these challenges will make the benefits of utility computing systems much more attainable – and this is a perfect time to do so, because utility computing systems are still at a sufficiently formative stage that there's a real chance to change them in fruitful ways.

Acknowledgments

Eric Anderson was the person who first suggested the problem to us. Discussions with several HP Labs colleagues helped crystallize our understanding of the space and clarify this paper.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P.Reynolds, and A. Muthitacharoen. *Performance Debugging for Distributed Systems of Black Boxes*, Proc. 19th ACM Symp. on Operating Systems Principles (SOSP'03), pp. 74–89, Oct. 2003

[2] M. Balazinska, H. Balakrishnan, and M. Stonebraker. *Contract-Based Load Management in Federated Distributed Systems*. Proc. 1st Symp. on Networked Systems Design and Implementation (NSDI), March 2004.

[3] P. Barham, R. Isaacs, R. Mortier, and D.Narayanan. *Magpie: real-time modelling and performance-aware systems*. 9th workshop on Hot Topics in Operating Systems (HotOS-IX), May 2003

[4] Cendura Corporation. *Intelligent application blueprints: managing modern distributed applications*, http://www.cendura.com/pdf/CohesionBrochure.pdf, 2004

[5] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, E. Brewer. *Path-Based Failure and Evolution Management.* Proc. 1st Symp. on Networked Systems Design and Implementation (NSDI), March 2004.

[6] Collation, Inc. *Collation Configia*, http://www.collation.com/products/, 2003.

[7] Hewlett-Packard. *The HP vision for the Adaptive Enterprise: achieving business agility.* http://www.hp.com/go/adaptive, part 5981-6177EN, July 2003.

[8] Hewlett-Packard. *HP Utility Data Center: transforming data center economics.* part 5982-3291EN, Mar. 2004.

[9] Hewlett-Packard. *HP Labs goes Hollywood.* http://www.hpl.hp.com/news/2004/apr-jun/nab.html, Apr. 2004.

[10] IBM. *On demand business.* http://www.ibm.com/e-business, 2004.

[11] M. Karlsson, C. Karamanolis and X. Zhu. *Triage: performance isolation and differentiation for storage systems*. Intl. Workshop on Quality of Service (IWQoS), pp. 67–74, June 2004.

[12] K. Keeton, C. Santos, D. Beyer, J. Chase and J. Wilkes. *Designing for disasters*. File and Storage Technologies (FAST'04), March-April 2004.

[13] *Utility computing*. IBM Systems Journal special issue **43**(1), 2004.

[14] Oracle Corporation. *Oracle database 10g*. http://otn.oracle.com/products/database/oracle10g, 2004.

[15] Sun Microsystems. *N1 Grid - Introducing Just In Time Computing*. http://wwws.sun.com/software/solutions/n1/wp-n1.pdf, 2002.

[16] J. Wilkes, P. Goldsack, J. Janakiraman, L. Russell, S. Singhal, and A. Thomas. *eOS - the dawn of the resource economy*. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001.

[17] J. Wilkes. *Traveling to Rome: QoS specifications for automated storage system management.* Intl. Workshop on Quality of Service (IWQoS'2001), pp. 75–91, June 2001. Published as Springer-Verlag Lecture Notes in Computer Science **2092**.

[2] We refuse to call this process *servicification*!