Exposed versus Encapsulated Approaches to Grid Service Architecture

Micah Beck, Terry Moore and James S. Plank
Center for Information Technology Research
University of Tennessee
Knoxville, TN 37996

Abstract — The dichotomy between the exposed and encapsulated approaches to computer systems architectures is well known in contexts such as the processor design (RISC vs. CISC) and layered network service stacks. In this paper we examine how this basic choice of approaches arises in the design of the Internet Backplane Protocol, a network storage service, and as an issue in Grid architecture more generally.

1. Introduction

One plausible interpretation of the progressive transformation of the Metacomputing of the last decade [1] into the "Grid" computing of this one [2] views it as an effort to realize the potential truth in the slogan that "the Network is the Computer," and to make that realization the basis of a public infrastructure for next generation scientific computing. A trio of powerful trends that dominated the 90's fueled this idea. First, the Internet, driven by the advent of the Web, experienced unprecedented growth, becoming an all-pervasive fabric of connectivity that application developers could assume to be present. Second, the rapid buildup of advanced research networks offered the possibility of guaranteed quality of service, end-to-end, for Internet applications on a national WAN. Finally, the continued exponential growth in the local provisions of all fundamental computing resources — processing power, communication bandwidth, and storage — suggested a picture of a network-as-computer of staggering aggregate capacity, if only the necessary software infrastructure could be created to bring these elements together.

But if the network is going to be the computer, the natural question is "What kind of computer is it going to be?" Or more directly, "What engineering approach should we take in building it?" In this paper we discuss what we believe to be a key architectural choice to be made in this endeavor, namely, the choice between an *encapsulated* and an *exposed* approach to building high-level functionality from low-level Grid resources.

The distinction between these two approaches is elementary. Any complex, shared computing system requires an architecture that will allow it to provide high performance services and yet be able to support new functionality to address previously unanticipated purposes as they arise. A common way to address this requirement is software layering. At the lowest level such systems are made up of physical resources that implement primitive functions with little protection between them. At higher levels, computing resources are represented as objects that can be much larger and more complex than primitive memory words, and operations defined on those objects are similarly much more complex than primitive machine instructions. What links the primitive and high levels is the *aggregation* of primitive memory and instructions to implement high level objects and operations. In an *encapsulated* approach to service architecture the necessary aggregation of low-level resources is hidden from the user at the higher level; in an *exposed* approach, the aggregation is external to the primitive service so that the low-level resource remains visible at higher levels.

This contrast between encapsulated and exposed approaches to resource engineering is widely known. Most notably, it appears in the historical debate between the supporters of *Complex Instruction Set Computers* (*CISC*) and the supporters of *Reduced Instruction Set Computers* (*RISC*) over how to make the best use of extra processor real estate [3]. Similarly, the decision in the late 70's to implement only the most essential and common communication functions at the network (IP) layer, forcing all stronger functionalities to build on that layer, represents a clear choice in favor of a more exposed approach to resource engineering for the Internet [4, 5].

One way to analyze the choice between exposed and encapsulated Grid architectures is to focus on the fact that, since the infrastructure must be shared by a large group of stakeholders, design approaches will tend to divide in terms of the way they structure that sharing. The *Computer Center model*, for example, which informs the current Grid paradigm, was developed in order to allow scarce and extremely valuable resources to be shared by a select community in an environment where security and accountability are major concerns. Consequently the form of sharing it implements is necessarily highly controlled [6] and access to low-level resources tends to be highly encapsulated. By contrast, the *Internet model* was designed to facilitate the sharing of network bandwidth for the purpose universal communication among an international community of indefinite size. It is therefore designed to be as open (i.e. lightly controlled) and easy to use as possible, and so it tends to leave low-level resources relatively exposed. While admission and accounting policies are difficult to implement in this model, the universality, generality, and scalability of the resource sharing it implements for communication bandwidth has obviously proved powerful.

Given the evident success of the Internet model, what seems most striking to us is that, as the research community continues to roll pell-mell toward a network-as-computer infrastructure, precious little research is being done to explore the possibility of taking a more exposed approach to the other basic elements of network computing, viz. *storage* and *computation*. In this paper we discuss some of the issues that have arisen during our efforts to investigate and build an exposed service for network storage. After filling out the contrast between encapsulated and exposed design philosophies for network services in general, we look in detail at two specific instances that have arisen during our work on distributed storage: one where the exposed approach seems to apply in a natural way (implementation of a file abstraction) and one where its application is less straightforward and more problematic (implementation of complex data movement operations). Our goal here is not to settle any question, but to open a conversation about an extremely important design option for the scientific computing community that is all but completely neglected at the moment.

2. Encapsulated vs. Exposed Network Services

To the extent that the scientific computing community is already using the network as a computer, the Internet provides a ubiquitous communication substrate connecting its components (with routers acting as special-purpose elements invisible in the architecture), while network servers provide all access to storage and computation. Illustrations of such servers and services are plentiful: FTP, NFS, and AFS [7] provide access to storage; Condor [8], NetSolve [9], Ninf [10] provide lightweight access to processing; HTTP provides access to both; GRAM [11] provides access to heavyweight computing resources; LDAP provides access to directory services; and so on. What is notable about these instances, and is equally true of almost all the other cases we could add to the list, is that they represent relatively *encapsulated* network services:

> An *encapsulated* network service implements functionality that does not closely model the underlying network resource, but which must be implemented by aggregating the resource and/or applying significant additional logic in its utilization.

The best effort delivery of datagrams at the IP level, on the other hand, represents a clear example of a relatively *exposed* network service:

> An *exposed* network service adds enough additional abstraction to the underlying network resource to allow it to be utilized at the next higher level, but does not aggregate it or add logic beyond what is necessary for the most common and indispensable functionality that uses it.

An important difference between the two approaches emerges when we need to extend the functionality of a given service. Encapsulated services tend to be implemented by heavyweight servers and have APIs designed at a high semantic level, interposing themselves between the client and low overhead, transparent access to the underlying resources As a result, it can be difficult, inefficient, or in some cases impossible to build new functionality on top of such APIs. Instead, encapsulated services tend to implement new functionality through "plug in modules" that extend the functionality of the server, introducing new code that has access to low level interfaces within the server. These plug-in modules are the server equivalent of microcode in CISC processors, raising a familiar set of questions about access control and security for the management of such code. Encapsulation also tends to lead to balkanization, with each server supporting a different set of plug-ins.

Extending the functionality of an exposed service makes different demands because exposed services have lighter weight servers and APIs designed at a simpler semantic level. Since these factors are conducive to lower overhead and more transparent access to the underlying resources, it tends to be much easier and more efficient to build new functionality on top of exposed services. Exposed services promote the layering of higher-level functionality on top of their APIs, either in higher-level servers or in client code.

This layering of services, which is analogous to the user-level scheduling of a RISC processor by a compiler, is perhaps most familiar in the construction of a network services stack. In the world of end-to-end packet delivery, it has long been understood that TCP, a protocol with strong semantic properties (e.g., reliability and in-order delivery) can be layered on top of IP, a weak datagram delivery mechanism. By allowing IP services to retain their weak semantics, and thereby *leaving the underlying communication bandwidth exposed for use by the broadest possible range of purposes*, this layering has had the crucial benefit of fostering ubiquitous deployment. At the same time, in spite of the weak properties of IP datagram delivery, stronger properties like reliability and in-order delivery of packets can be achieved through the fundamental mechanism of retransmitting IP packets. Retransmission controlled by a higher layer protocol, combined with protocol state maintained at the endpoints, overcomes non-delivery of packets. All non-transient conditions that interrupt the reliable, in-order flow of packets can then be reduced to non-delivery. We view retransmission as an *aggregation* of weak IP datagram delivery services to implement a stronger TCP connection.

Despite the familiarity of this exposed approach, it may still not be obvious how to apply it to a resource such as storage. After all, almost every technology for the access and/or management of network storage one can think of—FTP, HTTP, NFS, AFS, HPSS, GASS [12], SRB [13], NAS [14], etc.—encapsulates the storage behind abstractions with relatively strong semantic properties. For that reason, our research in this area had to start by creating a protocol, viz. the *Internet Backplane Protocol* (*IBP*), that supported the management of remote storage resources while leaving them as exposed as possible. IBP is a network service that provides an exposed abstraction of shared network storage [15, 16]. Each IBP *depot* (server) provides access to an underlying storage resource to any client that connects to the server. In order to enable sharing, the depot hides details such as disk addresses, and provides a very primitive capability-based mechanism to safeguard the integrity of data stored at the depot. IBP's low level, low overhead model of storage is designed to allow more complex structures, such as asynchronous networking primitives and file and database systems, to be built on top of the IBP API. The IBP depot and client library are now available for several Unix/Linux variants and Windows, and there is a Java client library (http://icl.cs.utk.edu/ibp); the API is documented in detail [17]. With IBP in place the question becomes how easy or difficult it is to layer storage services with strong semantic properties on top of the weak underlying storage resources provided by IBP depots.

3. Extending the Functionality of IBP

A key principle of exposed designs is that the semantics of low level services should be kept as weak as possible. To illustrate how weak the semantics of the IBP storage service is, we examine the primitive unit of IBP storage allocation, the *byte array*. As an abstraction the IBP byte array is at a higher level than the disk block (a fixed size byte array), and is implemented by aggregating disk blocks and using auxiliary data structures and algorithms. Abstracting away the size of the disk block, a byte array amortizes the overhead of allocation across multiple blocks. If we consider storage services at the disk block level to be the equivalent of "scalar" operations within a processor, then byte arrays allow a kind of "vectorization" of operations. Though our aim was to make the IBP storage service as exposed as possible, this level of encapsulation was considered indispensable to hide the most specific underlying characteristics of the access layer (physical medium and OS drivers) and to amortize per-operation overhead across multiple blocks.

Nonetheless, the semantics of the IPB byte array remain very primitive. This fact becomes clear when you realize that the most intuitive and universal abstraction for storage, viz. the *file*, has strong properties (e.g. unbounded size and duration of allocation) that are not generally available from the underlying storage resource and therefore are not modeled by IBP. Since abstractions with such strong and intuitive semantics are essential for ease of use, they must be implemented either in exposed style (by layering new functionality over the primitive service), or in encapsulated style (by adding powerful new operations that make the low-level service itself less primitive). Our experience has been that the former path is relatively easy to follow when implementing a file abstraction for exposed network storage, but that the latter is more straightforward for implementing a mechanism for one-to-many data movement.

3.1 Layering a file abstraction over IBP

In our exposed approach to network storage, the file abstraction must be implemented in a higher layer that aggregates more primitive IBP buffers. In order to apply the principle of aggregation to exposed storage services, it is necessary to maintain state that represents an aggregation of storage allocations, much as sequence numbers and timers are maintained to keep track of the state of a TCP session. Fortunately, in this case we have a traditional, well-understood model that can be followed. In the Unix file system, the data structure used to implement aggregation of underlying disk blocks is the *inode* (*intermediate node*). Under Unix a file is implemented as a tree of disk blocks with data blocks at the leaves. The intermediate nodes of this tree are the inodes, which are themselves stored on disk. The Unix inode implements only aggregation of disk blocks within a single disk volume to create large files; other strong properties are sometimes implemented through aggregation at a lower level (e.g. RAID) or through modifications to the file system or additional software layers that make redundant allocations and maintain additional state (e.g. AFS, HPSS) [7, 18].

Working by analogy with the inode, we have chosen to implement a single generalized data structure, which we call an *external node*, or *exNode*, for management of aggregate allocations that can be used in implementing network storage with many different strong semantic properties. Rather than aggregating blocks on a single disk volume, the exNode aggregates storage allocations on the Internet, and the exposed nature of IBP makes IBP storage allocations especially well adapted to such aggregations. In the present context the key point about the design of the exNode, which we describe in more detail elsewhere [19], is that it has allowed us to create an abstraction of a network file that can be layered over IBP-based storage in a way that is completely consistent with the exposed resource approach. In the case of one-to-many data movement, however, we have chosen to take a more encapsulated approach

3.2 One-to-many data movement using IBP

IBP's "vectorized" storage services work well in exposed mode as long as operations are simple. Likewise, the reliable transfer of data between client and depot, or between a source and destination depot, are a good fit if it is assumed that TCP can be implemented throughout the underlying network. But such is often not the case. The initial IBP API addressed point-to-point data movement with a single call that models the characteristics of transfer using a reliable TCP connection:

```
IBP_copy(source, target, size, offset)
```

The straightforward client API for IBP is not sufficiently exposed to allow for efficient transfer of data between a single source depot and multiple recipients: If one-to-many communication is implemented through repeated one-to-one communication, this communication can be optimized by reusing the source memory buffer rather than repeatedly reading from disk. Similarly, if there is an underlying UDP multicast service available, or if the network includes satellite, high performance or other links not conforming to the usual model of the public Internet, then TCP may not be the transport layer protocol of choice. However, complex signaling, flow control and retransmission may be required in order to implement reliable data transfer taking advantage of other transport layer protocols.

Given the need to manage low-level resources (e.g. memory buffers) when implementing one-to-many data movement, a natural approach is to implement such functionality at a low level. We have extended the current API with a more complex call that allows multiple targets and arbitrary data movement "operations" to be specified. These operations are implemented using low level processes called *datamovers* that plug into the depot software architecture between the depot process and the network

```
IBP_copy(DM_op, target_count, source, target[], int size, int offset)
```

By encapsulating data movement functionality at a level that allows for efficient access to underlying storage resources and can manage data transfer at the level of memory buffers, the software architecture finesses the problem of providing sufficient transparency to allow such functionality to be layered on top of the IBP API. In the process, however, the design has diverged from the philosophy of exposed-resource network services.

The exposed approach to this problem would seek to enable to implementation of complex data movement on top of the IBP API. You can see how this design approach would work by examining the problems that would arise in trying to use the current API and implementation to execute it:

- The current IBP protocol implements each operation as a separate TCP connection to the depot. However, this is easily optimized using persistent connections and multiplexing multiple concurrent operations over a single connection.
- The current IBP API does not model in-memory buffers and so cannot be used to explicitly optimize the movement of data between disk and network. The addition of short-lived memory buffers is an easy extension to the IBP API and would address this problem.
- High performance vectorized (multi-buffer) transfers of data require the immediate execution of each operation as soon as possible after its predecessor operations have completed. Timeouts and other complex strategies must be used to deal with exceptional conditions.

As in any architecture where there is a substantial latency between operation issue and execution, latency will also be a potential problem here. In processor architecture, the solutions to this problem include pipelining with built-in interlocks to implement dependences, and both predicated and speculative execution. An exposed approach to enhancing the data movement functionality of the IBP depot would follow these strategies. The result would be a more complex, but highly general operation-scheduling function that allows stronger operations to be implemented at a higher level.

4. Conclusions

Starting with any initial design, there is always the temptation to extend it by adding new operations and encapsulating their implementation in the bowels of the existing system. This often has the advantage of backward compatibility with the existing service and of providing maximum control in the implementation of the new functionality. We can see this approach being followed in the addition of DataMovers to the IBP storage service: plug-in functionality was added at the low level and a call directly invoking that new functionality was added to the client API.

Theexposedresourceapproachtonetworkservicesfollowsinalongtraditionofhardwareandsoftware architecture.When,asinthecaseoftheexNode,itisfeasibletolayernewfunctionalityoveranexistingAPI, thatisusuallythepreferredapproach.However,whenimplementingnewfunctionalityrequiresthattheservice bemodifiedtoexposenewresourcesandprovidemorepowerfulschedulingfunctions,takingtheexposed approachposesseveralrisks:

- Theexistingservicemightneedtobesubstantiallyrestructuredtosupportthenewfunctionalityata higherlayer.
- Theexposeddesignofthelowerlayermightbetoocomplextobeeasilyprogrammedatthehigher layer.
- Theperformanceofthelayeredimplementationmaybeinadequateduetotheoverheadofmaking multiplecallstothelowerlayer.
- Inadequateanalysisoftherequirementsforaccesstolowlevelresourcesmayresultinanexposed servicethatcannotsupportthenewfunctionalityatahigherlayer.

Theserisksseemedsubstantialenoughtoleadustotaketheencapsulatedapproachinthecurrent implementationofone-to-manydatamovementusingIBP.However,thedesignofanexposeddatamovement interfaceisbeingstudied,andaprototypeimplementationisplanned.Thepossiblebenefitsofanexposed resourceapproachtoGridservicearchitecturearetoogreattoleaveunexplored.

Ifthenetworkisgoingtobethecomputer,andifexposedapproachestonetworkingandstoragecanbe developed,thenthefinalcomponentiscomputation.Exposedapproachestocomputationwouldrequirethatthe processorresourcebeexposedtothenetworkinarawform,allowingarbitrarycomputationstobeperformed upondatastoredinanexposedstorageserviceandtransportedusingexposednetworkingservices.Several differentGridelementsinthecurrentensembleofservices,includingGRAMandNetworkEnabledServers suchasNetSolveandNinf,performcomputationsforremoteclients.Inthesecasesthereisatradeoffbetween opennessandgenerality:Gramwillexecutearbitrarycodeforaknownuser;NetSolveandNinfwillperform computationsonbehalfofarbitraryusersbutwillexecuteonlyknownandtrustedcode.Thedevelopmentofa networkservicethatstrikesabalancebetweenopennessandgenerality,whichwecallthe *NetworkFunctional Unit*,wouldbetheultimateandmostdifficultachievementoftheexposedapproachtoGridservicearchitecture.

5. References

[1] L.SmarrandC.E.Catlett,"Metacomputing," *CommunicationsoftheACM*,vol.35,no.6,pp.44-52, 1992.

[2] I.FosterandC.Kesselman,"TheGrid:BlueprintforaNewComputingInfrastructure."SanFrancisco, CA:MorganKaufmanPublishers,1998,pp.677.

[3] J.L.HennessyandD.A.Patterson, *ComputerArchitecture:AQuantitativeApproach*,2nded.San Francisco,CA.:MorganKaufmannPublishers,Inc.,1996.

[4] J.H.Saltzer,D.P.Reed,andD.D.Clark,"End-to-EndArgumentsinSystemDesign," *ACM TransactionsonComputerSystems*,vol.2,no.4,pp.277-288,1984.

[5] D.P.Reed,J.H.Saltzer,andD.D.Clark,"CommentonActiveNetworkingandEnd-to-End Arguments," *IEEENetwork*,vol.12,no.3,pp.69-71,1998.

[6] I.Foster,C.Kesselman,andS.Tuecke,"TheAnatomyoftheGrid:EnablingScalableVirtual Organizations," *InternationalJournalofSuperComputerApplications*,2001.

[7] J.H.Morris,M.Satyanarayan,M.H.Conner,J.H.Howard,D.S.H.Rosenthal,andF.D.Smith, "Andrew:ADistributedPersonalComputingEnvironment," *CommunicationsoftheACM*,vol.29,no. 3,pp.184-201,1986.

[8] J.PruyneandM.Livny,"Aworldwideflockofcondors:Loadsharingamongworkstationclusters," *JournalonFutureGenerationsofComputerSystems*,vol.12,1996.

[9] H.CasanovaandJ.Dongarra,"ApplyingNetSolve'sNetworkEnabledServer," *IEEEComputational Science&Engineering*,vol.5,no.3,pp.57-66,1998.

[10] S.Sekiguchi,M.Sato,H.Nakada,S.Matsuoka,andU.Nagashima,"Ninf:NetworkbasedInformation LibraryforGloballyHighPerformanceComputing,"presentedatProc.ofParallelObject-Oriented MethodsandApplications(POOMA),SantaFe,NM,1996.

[11] I.Foster,C.Kesselman,C.Lee,R.Lindell,K. Nahrstedt,andA.Roy,"ADistributedResource ManagementArchitecturethatSupportsAdvanceReservationsandCo-Allocation,"presentedatIntl WorkshoponQualityofService,1999.

[12] J.Bester,I.Foster,C.Kesselman,J.Tedesco, andS.Tuecke,"GASS:ADataMovementandAccess ServiceforWideAreaComputingSystems,"presented atSixthWorkshoponI/OinParalleland DistributedSystems,May5,1999,1999.

[13] C.Baru,R.Moore,A.Rajasekar,andM.Wan," TheSDSCStorageResourceBroker,"presentedat CASCON'98,Toronto,Canada,1998.

[14] G.GibsonandR.V.Meter,"NetworkAttachedStorageArchitecture," *CommunicationsoftheACM* , vol.43,no.11,pp.37-45,2000.

[15] J.Plank,M.Beck,W.Elwasif,T.Moore,M.Swany,andR.Wolski,"TheInternetBackplaneProtocol: StorageintheNetwork,"presentedatNetStore99:TheNetworkStorageSymposium,Seattle,WA, 1999.

[16] M.Beck,T.Moore,J.Plank,andM.Swany,"LogisticalNetworking:SharingMoreThantheWires," in *ActiveMiddlewareServices* ,vol.583, *TheKluwerInternationalSeriesinEngineeringand ComputerScience* ,S.Hariri,C.Lee,andC.Raghavendra,Eds.Boston: KluwerAcademicPublishers, 2000.

[17] A.Bassi,M.Beck,J.Plank,andR.Wolski,"InternetBackplaneProtocol:API1.0,"Departmentof ComputerScience,UniversityofTennessee,Knoxville,TN,CSTechnicalReport,ut-cs-01-455,March 16,20012001.http://www.cs.utk.edu/~library/2001.html.

[18] R.W.WatsonandR.A.Coyne,"TheParallelI/O ArchitectureoftheHigh-PerformanceStorage System(HPSS),"presentedatIEEEMassStorageSystems Symposium,1995.

[19] A.Bassi,M.Beck,andT.Moore,"MobileManagementofNetworkFiles,"in *ThirdAnnual InternationalWorkshoponActiveMiddlewareServices*.SanFranscisco,2001,toappear.