TRANSPARENCY IN ITS PLACE -- The Case Against Transparent Access to Geographically Distributed Data

Jim Gray

Tandem Computers Inc.
19333 Vallco Parkway
Cupertino Ca

Tandem Technical Report TR89.1
Tandem Part Number PN 21667

TRANSPARENCY IN ITS PLACE --
The Case Against Transparent Access to Geographically Distributed Data

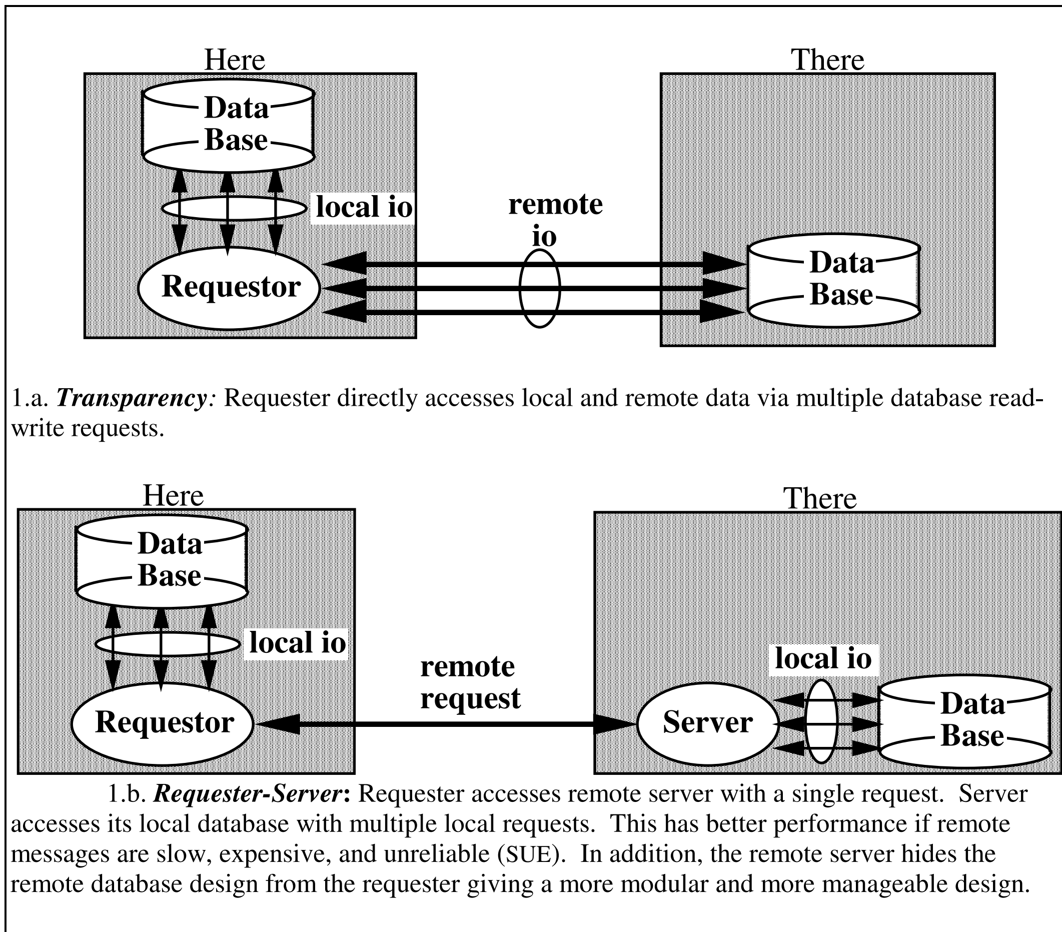
Jim Gray
Tandem Computers Inc.
19333 Vallco Parkway
Cupertino Ca
May 1987+
Revised Feb. 1989

Abstract: Distributed database software offers *transparent* access to data -- no matter where in the network the data is located, an authorized program can access the data as though it is local. This article argues that transparency in a geographically distributed system is unmanageable and has technical drawbacks. The real virtue of transparency is its ability to support geographically centralized clusters of computers.

+ This paper originally appeared in UNIX Review, V.5, N.2, May 1987, pp. 42-50.

Distributed database software offers *transparent* access to data -- no matter where in the network the data is located, an authorized program can access the data as though it is local. Transparency has been the goal of distributed database systems for over a decade -- it is at the core of next-generation distributed database systems.

Both IBM's CICS/ISC [1] and Tandem's Encompass [2] have offered transparency since 1976. Although these two distributed systems are very popular, their ability to transparently access remote data is not used much. In fact, design experts of both vendors recommend against transparent remote access to data [3]; instead, they recommend a requester-server design (sometimes called remote procedure call) in which requests for remote data are sent to a server which accesses its local data to service the request (see figure 1). Ironically, CICS and Encompass initially offered only transparent distributed data. Both added a requester-server model about three years later (1979).



1.a. ***Transparency**:* Requester directly accesses local and remote data via multiple database read-write requests.

1.b. ***Requester-Server*:** Requester accesses remote server with a single request. Server accesses its local database with multiple local requests. This has better performance if remote messages are slow, expensive, and unreliable (SUE). In addition, the remote server hides the remote database design from the requester giving a more modular and more manageable design.

Why are newcomers enthusiastic about the transparency they will deliver next year, while old-timers who have had these facilities for a decade are skeptical about using transparency for geographically distributed applications? Manageability is at the crux of this paradox. Geographically distributed system are a nightmare to operate. By definition, they are a large and complex system involving hundreds if not thousands of people. Communication among computers, administrators and operators in such a system is slow, unreliable, and expensive (SUE for short). When confronted with a "real" distributed system, one with SUE communication, successful application designers fall back on the requester-server model. Such designs minimize communication among sites and maximize modularity and local autonomy for each site.

Distributed database systems do have an important application: they allow modular growth -- the ability to grow a system by adding hardware modules to a cluster instead of growing by trading up to a bigger, more expensive box (see Figure 2). It is not enough to have a "cluster" hardware architecture -- distributed system software is needed to allow modular growth. Clusters avoid all the hard problems of geographic distribution:

- Clusters have high-bandwidth, low-latency, reliable and cheap communication among all processors in the cluster. All data in the cluster is "close" since access times are dominated by disk access time. Geographically distributed systems must deal with slow, unreliable, expensive communications via public networks. In geographically distributed systems message delays dominate access times.
- A cluster can be administered and operated by a single group with face-to-face contact and with similar organizational goals. "Real" distributed systems involve multiple sites, each site having its own administration. Personal communication and relations among sites are via slow, unreliable, expensive mail and telephones -- SUE again.

This analysis is controversial. But, if it's correct, computer vendors should overhaul their hardware and software to add support for clusters. On the other hand, there is little need for computer users to worry about geographically distributed or heterogeneous databases -- rather a standard requester-server model is needed so that application designers can build geographically distributed systems in standard ways. I believe that IBM's SNA LU6.2 will emerge as that standard [4]. In addition, standard data interchange protocols are needed. Facilities being built atop LU6.2 are likely to become those standards.

The rest of this article attempts to justify this analysis and its conclusions.

FIGURE 2.a: The classic **VonNeumann machine** with a Central Processing Unit (CPU) that handles both computation and IO. Virtually no one builds such a computer today.

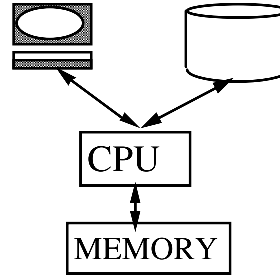


Figure 2.b: The first evolutionary step away from the VonNeumann model: **multiple processors share memory**. Some of the processors are functionally specialized to do IO while others execute programs. The processors communicate via shared memory and signal wires.

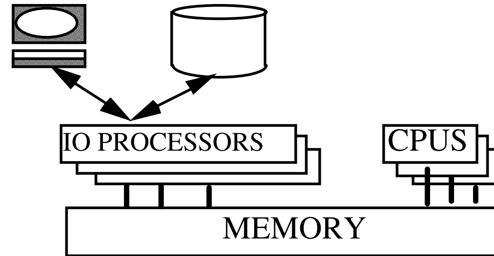


Figure 2.c: The next evolutionary step was to couple multiple independent processors on a high speed local network to form a **cluster**. The nodes of the cluster do not share memory, rather they communicate via messages. This architecture allows tens or hundreds of processors to be applied to one application. It is the thesis of this article that clusters require a distributed database system to be effective and distributed database transparency should only be used within a cluster.

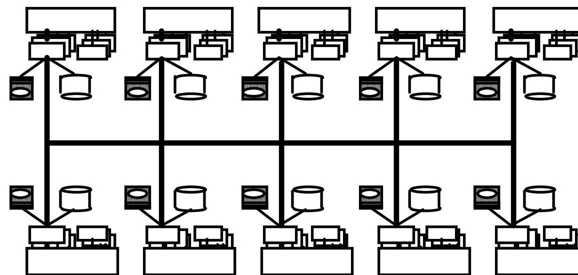


Figure 2.d: A true distributed system in which **geographically distributed clusters** are connected via long-haul networks. As with a cluster, the nodes do not share memory, rather they communicate via messages. But in this case messages are Slow, Unreliable and Expensive (SUE). This architecture allows geographically distributed applications and data. It is the thesis of this article that truly distributed systems require a requester-server mechanism to be effective -- distributed database transparency should not be used in a long-haul network.

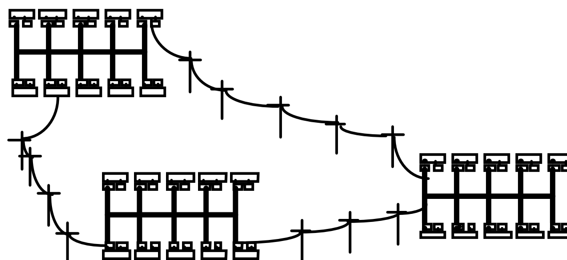


Figure 2: The evolution of computer architectures to clusters and networks.

The Case Against Transparent Access to Distributed Data

Distributed databases offer transparent access to data. Beyond the authorization mechanism, there is no control over what the program does to the data. It can delete all the data, zero it, or insert random new data. In addition, no comprehensible audit trail is kept telling who did what to the data. This interface is convenient for programmers, but it is a real problem for application designers and administrators.

The simplest way to explain the negative aspects of a distributed database is to compare refrigerators to grocery stores. My refrigerator operates like a distributed database. Anyone with a key to my house is welcome to take things from the refrigerator or put them in. There is a rule that whoever takes the last beer should get more at the grocery store. I only give keys to people who follow this rule.

A grocery store could operate like a distributed database. It could hand out keys to trusted customers who agree to pay for any groceries they take. This would be much cheaper than having a lot of clerks standing around collecting money from customers. Why don't any grocery stores operate in this way? Why are they different from refrigerators? Well, its because refrigerators are convenient for the users but are unmanageable. The clerks manage the access to the store inventory.

Requester-server designs provide an administrative mechanism much like the store clerks. They provide defined, enforceable, auditable interfaces which control access to an organization's data. Rather than publishing its database design and providing transparent access to it, an organization publishes the CALL and RETURN messages of its server procedures. These servers perform requests according to the procedures specified by the site owner. They are the site's standard operating procedures. Requesters send messages to servers which in turn execute these procedures much as the clerks perform and enforce the store's operating procedures.

Requester-server designs are more modular than distributed databases. A site can change its database design and operating procedures without impacting any requesters. This gives each site considerable local autonomy. The only things a

site cannot easily change are the request and reply message formats. In the parlance of programming languages, distributed databases offer transparent types, servers offer opaque types, sometimes called abstract types or encapsulated types.

Requester-server designs are more efficient. They send fewer and shorter messages. Consider the example of adding an invoice to a remote node's database. A distributed database implementation would send an update to the account file, insert a record in the invoice file, and then insert several records in the invoice-detail file. This would add up to a dozen or more messages. A requester-server design would send a single message to a server. The server would then perform the updates as local operations (see Figure 1). If the communication net is SUE then sending only a single message is a big savings over multi-message designs.

To summarize the negatives, applications coded with transparent access to geographically distributed databases have:

- Poor manageability,
- Poor modularity and,
- Poor message performance.

when compared to a requester-server design.

The lunatic fringe of distributed databases promise transparent access to heterogeneous databases (say an ASCII system accessing an EBCIDIC system). These folks promise to hide all the nasties of networking, security, performance, and semantics under the veil of transparency.

This is a wonderful promise. But the prospect of getting people who cannot agree on how to represent the letter "A" to agree to share their raw data is far fetched. Heterogeneous systems are a very good argument for requesters and servers. The systems need only agree on a network protocol and a requester-server interface. Still a little far fetched unless a standard network and requester-server model emerges.

Manageability of a Distributed Database

Manageability is the key problem in distributed systems. Lets suppose for the moment that some genius solved all the technical problems. Lets suppose that there is cheap, fast, reliable communications among all points of the globe. Suppose that everyone agrees to run the same hardware and same software. Suppose that everyone trusts everyone else completely and that there are no auditors insisting that we explain how our system works.

Now suppose that we have to design and manage a distributed application in this ideal world. Will we use transparent access to geographically remote data? Probably not.

Why not? Well, a distributed system is a big and complex thing. We will want to change and grow it over time. We may want to add nodes, move data about, redesign the database, change the format or meaning of certain data items, and do other things which are likely to invalidate some programs using the data.

If everyone in the world knows what our database looks like, and we change the design, then their programs will stop working. Some changes may not break programs, but others certainly will. To install a change, we would have to change all the programs that use our data. This might be possible, but at some point, change control will consume all the system's resources.

Modularity is the solution to this. If we only tell people about the interface to our servers, we can change a lot about our database without letting anyone else know. We can support "old" server interfaces when we go to a new design and gradually inform our users about the new interface. They can convert at their leisure.

So, even in the programmer's ideal world, manageable distributed systems must be structured as modules communicating via messages rather than as programs transparently accessing an integrated distributed database.

The Case For Transparent Access to Cluster Data

What is the proper place for transparent access to distributed databases? Transparent access is very convenient for programmers -- it makes it easy to bring up distributed applications. Coding requesters-servers and making an application modular is extra work. Unfortunately, system administrators control the security of their data and generally do everything in their power to prevent ad hoc queries from running on it. If I want to run an ad hoc query on someone else's data, I have to get permission. This turns out to be not very ad hoc after all.

Clustering is the real application of transparent access to distributed data. To appreciate clusters, you have to appreciate the quandary of computer vendors. Almost all vendors have standardized on a single architecture. IBM wishes it had only System 360, DEC wishes it had only VAX and so on. The vendors then build 1, 2, 3, 4, 5,... MIP engines for that architecture. Most vendors are limited to 15MIPs per processor right now. To go beyond that they must combine several processors and convince the customer that the resulting price/performance adds to more than 15MIPS.

Clusters offer an approach to this problem. The vendor builds a slow-cheap cpu (say 1MIP), and a fast-expensive cpu (say 10MIPS). The vendor does the same for disks and communication controllers -- making a cheap box and a high-performance box. He then offers software that lets the customer use between 1 and 100 processors clustered as a single system. This gives the customer a 1MIP to 100MIP range with the cheap engines and a 10MIP to 1000MIP range for the expensive boxes (see figure 3).

Clustering offers both the customer and the vendor significant advantages. The customer can buy just what he needs and grow in small increments as he needs more. The vendor has two advantages. First it need design and support only a very few module types (discs, cpus, communications,....). In addition, it can build systems which far exceed the power of the non-clustered vendors. Apollo, DEC, Teradata, Tandem, and Sun have each taken this approach. Of course if the vendor or customer programs in a bottleneck, then the clusters cannot grow beyond the bottleneck. Successful vendors and customers have avoided such bottlenecks -- it is possible but the many failures indicate that it is not easy.

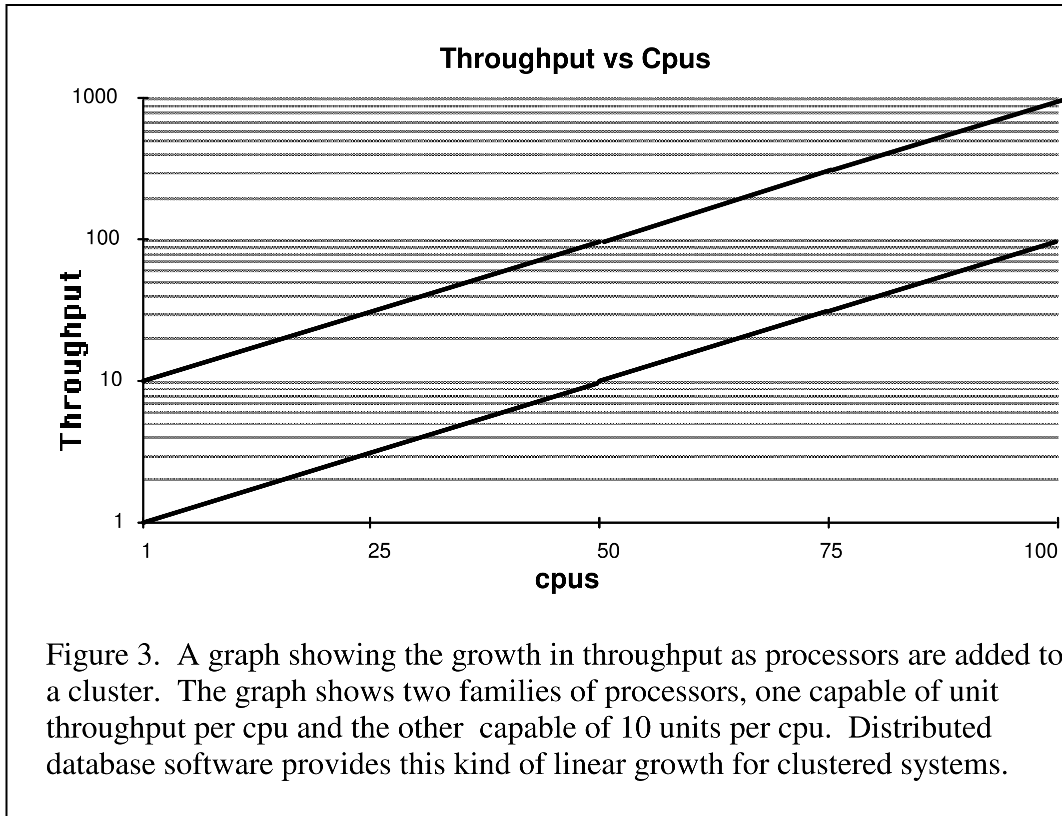


Figure 3. A graph showing the growth in throughput as processors are added to a cluster. The graph shows two families of processors, one capable of unit throughput per cpu and the other capable of 10 units per cpu. Distributed database software provides this kind of linear growth for clustered systems.

The arguments against geographically distributed databases do not apply to clustered systems. A cluster and its operators are typically in a single room. SUE is not a problem. The people have face-to-face contact and the computers have duplexed, high-speed buses among them.

A cluster is like a centralized system, so it can be managed as one. For small clusters the local autonomy derived from modularity may be moot. Reviewers of this paper took strong exception to that statement. They argue that any cluster which supports several applications will operate as a requester-server system just to enforce the modularity. Large applications must be decomposed into independent subsystems each of which is managed independently. The centralized cluster example in [3] is actually managed as five cooperating applications each with its own server interfaces to the others.

A distributed database serves cluster applications nicely, allowing data to be partitioned among any disks in the cluster and allowing servers to run on any cpus in the cluster. Because the intra-cluster communication is fast and cheap, the cost of distributing data in the cluster is negligible -- well perhaps not negligible but at least acceptable. Based Tandem's experience, the message-based design required for clustered systems uses about twice as many data moves and instructions as a "conventional" design. So clustered systems "waste" about half the MIPS in order to get a software design that supports modular growth within a cluster without bottlenecks. Tandem did this because they offer mirrored disks, duplexed data

paths and so on for fault tolerance -- other vendors have been reluctant to sacrifice a factor of two. And yet, Tandem and Teradata systems are competitive with those of other vendors and they are the only vendors building functional 100MIP clusters.

Conclusions

To summarize, the benefits of transparent access to clustered databases are undersold within computer vendors; and the benefits of transparent access to geographically distributed databases are oversold to customers. Customers wanting to implement geographically distributed applications need a standard and powerful requester-server mechanism.

It is fine to distribute data geographically close to the data users. But when access to the data crosses geographic or organizational lines, then the access is best structured as a requester-server interaction with a remote server which in turn accesses and updates its local data.

We hear relatively little about the requester-server idea -- there are no conferences or journals dedicated to it. Nonetheless, there is considerable activity in this area. Remote procedure call protocols typified by Courier from Xerox [5] and RPC from SUN [6] are being promoted. Tandem continues to extend its Pathway system [2]. IBM is implementing SNA LU6.2 which is the Esperanto of the IBM data processing world [6].

SNA LU6.2, also known as Advanced Program to Program Communication (APPC), is the de facto standard requester-server mechanism. All the major vendors have announced their intention to support it. In addition, IBM is building an edifice of software atop LU6.2 including transaction processing (CICS), name servers (SNADS), distributed database (DDM and CICS), forms flow and electronic mail (DISOS), document interchange (DIA/DCA), and so on. These extensions are servers defined by the server's message formats and English specifications of the server's semantics.

LU6.2 is a set of protocols to:

- Establish an authorized and authenticated conversation between a requester and a server which allows multi-message exchanges (calls) and informs the endpoints if the conversation or other endpoint fails.
- Exchange requests and replies between requesters and servers.
- Package a set of requests to a set of servers as a single atomic transaction so that all servers within the transaction will commit or all will undo their operations.
- Deal with errors along the way.

So LU6.2 is merely a remote procedure call mechanism combined with a transaction mechanism (a transaction commit/abort protocol) and an authorization mechanism. It gains it's significance from the 35,000 CICS systems (which all support it), the IBM System 38 which supports it nicely, the many applications that are being built on top of it, and the many non-IBM systems which have varying degrees of support.

It is likely that particular industries will evolve standard servers based either on LU6.2 or on higher level functions such as SNADS and DISOS. The universal support of LU6.2 and the support of industry-specific extensions are likely to solve many of the heterogeneous systems problems currently plaguing designers of distributed applications.

References

[1] *CICS/OS/VS Intercommunication Facilities Guide*, International Business Machines, White Plains, N.Y, Form SC33-0230, 1986.

[2] *An Introduction to Pathway*, Tandem Computers Inc. Cupertino, CA., Part: T82339, 1985.

[3] Anderton, M., Gray, J., "Four Case Studies of Distributed Systems", Tandem Computers Inc., Cupertino, CA. TR. 86.5 (1986).

[4] *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, International Business Machines, White Plains, N.Y, Form GC30-3084, 1986.

[5] "The Remote Procedure Call Protocol", Xerox Corp., Rochester NY, TR XSIS 038112, (1981).

[6] "Remote Procedure Call Specification", Sun Microsystems Inc., Sunnyvale, CA., (1985).