Tapping the Matrix

Carlos Justiniano
cjus@chessbrain.net, cjus34@yahoo.com

This paper was first published on the O'Reilly OpenP2P.com website (http://www.openp2p.com/) as a two part article in April 2004.

Neo awakes amidst a vast field of towering pods. Each pod, connected to the matrix, contains a person whose bioelectric energy is being harvested as fuel for a race of machines. That was one of the early scenes in the blockbuster movie The Matrix. The concept, although a stretch of the imagination, has an inverted parallel -- humans are harvesting the processing power of the millions of machines connected to a matrix we call the Internet.

This opportunity is made possible, in part, by the realization that a staggering number of computers are vastly underutilized. This squandering of resources doesn't just involve home computers; few businesses utilize their computers the full 24 hours of any day. In fact, ubiquitous applications, such as word processing, email, and web browsing, require very few CPU resources to support their end users, so machines are largely idle even when they appear to be in use. Just think about what your computer is doing right now, as you read this article.

Modern machines are capable of executing a billion instructions in the time it takes us to blink. The fact is that many computers today spend a significant amount of their time displaying multicolored swirls generated by screen saver programs. This is in striking contrast to the golden age of the mainframe, when time-sharing systems demanded a premium for their precious CPU cycles.

This situation has not gone unnoticed. For researchers, the lure of harnessing spare computing cycles has been simply too good to pass up. The benefits, potentially faster computation at substantially lower costs, are made possible because the bulk of the overall computing power comes from remote machines -- machines that are individually owned and operated by the general public.

Distributed Computing, Clusters, Grid Computing, Parallel Computing, and P2P -- You Are Here!

Pop quiz: how do you create a really fast computer? You basically have two choices.

1. Create faster processors and components.
2. Use more than one processor.

Since the late 1970s, some of the fastest computers have been built using multiple processors. The goal has been to speed up execution by allowing programs to execute on multiple processors while sharing resources such as memory. This has also allowed programs to communicate with one another as they collaborate on a particular problem. This divide-and-conquer approach to computing is known as parallel computing and is fundamentally how complex problems are solved using supercomputers. The tight integration of specialized processors, memory, and the hardware to effectively connect components produces extremely fast computation, but also accounts for the exorbitant costs of supercomputers.

During the past decade, personal computers have matured into fast and capable machines. Researchers have explored ways of connecting groups of machines to form clusters capable of rivaling larger, more expensive systems. Just ten years ago, NASA researchers Thomas Sterling and Don Becker connected 16 machines to create a cluster they called Beowulf. Today, Beowulf clusters involving thousands of computers are commonplace. In fact, Google claims to operate the world's largest Linux cluster involving more than ten thousand servers.

As you can imagine, monitoring thousands of computers can be a nightmare. To address this issue, specialized software must be built to manage clusters. This is where grid computing comes in. Grid computing software treats a collection of machines as resources that can be partitioned, allocated, and monitored according to established usage guidelines. In addition, grid computing platforms provide developers with methods of building software for use in grid environments.

At the time when clusters were emerging, developers began building software that allowed personal computers to share resources by allowing machines to loosely bind to one another to form highly dynamic networks. The new networks became known as Peer-to-Peer (P2P) networks, and were initially used for instant messaging (ICQ) and later, file sharing (Napster).

The emergence of P2P networks reminded researchers that in addition to instant messaging and file sharing, computation could also be shared. As a result, an increasing number of projects have been formed that are classified under the banner of distributed computing, and more

specifically within a subcategory known as distributed computation. We will use the term "distributed computation" throughout this article to refer to distributed computing projects, which specifically involve geographically dispersed machines all working to solve a computational problem.

The past decade as been incredibly fast-paced, and a significant cross-pollination of ideas has occurred. We're currently seeing P2P evolve to include grid-like capabilities and grids evolving to assimilate all things distributed computing.

Popular Projects

The O'Reilly Open Peer-to-Peer Directory and The AspenLeaf DC web site currently list several dozen active projects, ranging from analytical spectroscopy to the Zeta grid, specializing in areas such as art, cryptography, life sciences, mathematics, and game research.

One such project, Folding@Home, set out to use large-scale distributed computing to create a simulation involving the folding of proteins. Proteins, which are described as nature's nanobots, assemble themselves in a process known as folding. When proteins fail to fold correctly, the side effects include diseases such as Alzheimer's and Parkinson's.

The Folding@Home team described their protein simulation in the form of a computational model suitable for distribution. The result is now a tool that allows them to simulate folding using timescales that are thousands to millions of times longer than were previously possible.

Perhaps the most famous distributed computation project is SETI@home (the search for extraterrestrial intelligence) at the University of California in Berkeley. SETI@home distributes data collected from the radio telescope on the island of Puerto Rico to millions of remote machines throughout the world, where the data is analyzed for potential evidence of extraterrestrial civilizations.

SETI@home has demonstrated the viability of harnessing remote machines and has captured the imagination of over four million people, and has easily become the largest distributed computation project in history.

It All Begins with a Problem

Distributed computation projects work under the basic premise that machines working in parallel are able to solve a certain class of problems more cost effectively than a local group of machines.

If you consider starting a distributed computation project, one of the first challenges you are likely to encounter involves determining whether your particular problem is suitable for distribution. That is, can your problem be subdivided into discrete chunks? Furthermore, can the problem be subdivided so that there are no immediate interdependencies?

To help address this problem, you should consider writing programs to test the decomposition, distribution, collection, and assembly of processed work units. You have considerable flexibility in the choice of computer languages and other tools, which you can use to build quick prototypes. It is important not to get bogged down with too many technology-related issues. The key is to keep it simple -- work with data files and command-line programs and leave the actual inter-process and network communication aspects for a later time. The goals at this stage should focus on exploration and discovery, with an emphasis on the actual problem at hand.

Once you are confident that your problem can indeed benefit from distributed computation, your next step is to consider a general architecture for the various programs you'll need.

A Common Approach

At a fundamental level, a large majority of distributed computation projects work similarly. Let's examine a typical project from a 30-thousand-foot view:

> A central computer subdivides a problem into millions of smaller tasks. It then proceeds to dispatch each task to a remote machine. As each machine completes its assigned task, it returns a result back to the central computer, which may further subdivide (or create new) tasks. The process continues until a solution is found or all subtasks have been processed.

This process, while overly simplified, does reveal that a central entity dispatches work to remote entities, which later return results. For the sake of simplicity, I'll refer to the central computer as a "SuperNode," the remote machines as "PeerNodes," and subtasks as "work units." Note, however, that a SuperNode may consist of several physical servers, such as a scheduling server, database server, and other intermediate servers, to help balance network load.

You may have realized that the relationship between a PeerNode and SuperNode bares a striking resemblance to the relationship between web browsers and web servers. This similarity is no coincidence, because both have client/server attributes. We will examine this relationship later in this article; for now, keep in mind that PeerNodes can be considerably more dynamic than any modern-day web browser. Also, distributed computation projects are not restricted to client/server relationships and one-to-many network topologies. Indeed, some projects support the presence of multiple SuperNodes, which in turn cluster communities of PeerNodes. In addition, DC projects such as

Electric Sheep are exploring the use of P2P networks using protocols such as Gnutella.

Tim Toady. Tim Who?

The Perl programming community has a motto, "There's More Than One Way To Do It," which exemplifies the notion that a program is correct so long as it accomplishes its goal. Despite being rather long, the motto rings true in the distributed computing world, where we have an abundance of technological choices.

At one extreme, we have the all-encompassing grid frameworks, and at another we have the necessary raw materials (tools) to "build it ourselves." In between, we have proprietary frameworks such as Microsoft's .NET, various SUN Java frameworks, and open source tools such as LAMP (Linux, Apache, MySQL, Perl/PHP/Python). We also have several dozen readymade frameworks, such as the Berkeley Open Infrastructure for Network Computing (BOINC), which builds on open source LAMP tools); and Alchemi (A . NET-based Grid Computing Framework and its Integration into Global Grids), which leans toward the grid computing.

Which tools are best for you? Well, that depends on your own experience base and the unique requirements of your project. We'll continue exploring key issues so that you'll have a sense of how various tools work, and what you should consider when evaluating existing frameworks or before embarking on your own unique and custom solution.

Client/Server Communication

One of your most important considerations is how your project's primary servers will communicate with remote client machines. Again, we'll simplify our discussion by referring to one or more central servers as a single SuperNode, and multiple distributed clients as PeerNodes.

Consider the following questions. If you don't understand a question, ask a friend or simply enter the highlighted words into your favorite search engine.

- Will the communication need to be synchronous or asynchronous?
- Will the application communicate through port 80 or will it require another port?
- Will the SuperNode need to maintain sessions, or will transactions be stateless?
- Will PeerNode clients need to communicate through intermediary firewalls and proxy servers?
- How important is network transmission speed to the application?
- How large is the data intended for transmission to clients? How large will the return result be? Should the data be compressed?
- How important is data security? Does the data have to be encrypted at the server, the client, or both?
- Will PeerNodes need to communicate with one another?

Those communication questions can help you to carefully consider your project's specific goals and requirements, in addition to helping you evaluate the use of toolkits and frameworks.

In the end, your answers to those questions may lead you down the same path taken by many distributed computing projects. Specifically, toward the use of existing open standards, well-established web-based communication methodologies, and in particular, the use of the HTTP protocol. While you are free to explore alternative approaches (and you should, by all means), we'll focus on the most widely accepted methodologies. This will allow you to at least have a frame of reference as you consider the benefits of other approaches.

HTTP, XML, and SOAP?

The Hypertext Transfer Protocol (HTTP) is the underlying language that web servers and web browsers speak as they negotiate the transfer of web content. HTTP is a well-documented, well-tested text-based protocol that uses simple verbs such as GET and POST to specify actions. A core benefit of HTTP is that it includes support for interacting with other network devices such as routers, load balancers, and proxy cache servers. Once you consider the widespread deployment of web servers, web browsers, and devices that understand HTTP, it is easy to see why HTTP has become a standard for building distributed applications.

A common criticism of HTTP is that the size of your data may be smaller than an actual HTTP header. For example, an HTTP header may be 200 bytes long, while your actual data payload may be only 100 bytes in size. So it costs more to transmit the HTTP header than it does to send your unique application data. Depending on your choice of tools, you can control the size of the HTTP headers. However, you may reach a point of diminishing returns where you may not be able to significantly balance the header-to-payload ratio. Keep in mind that interoperability often requires compromise, and we'll continue to explore this concept later in this article.

Using HTTP verbs, software can send and request packaged data with no predefined limits on size or format. Indeed, Internet radio stations use HTTP to broadcast continuous streams of binary data on an open connection. The freedom to transmit any type of data further makes HTTP ideal; however, there is value in considering a structured approach to your HTTP payload data that will help enable interoperability. The XML markup language was created to facilitate the interchange of structured data by

allowing developers to use agreed-upon keywords. Consider the XML text below, which defines one or more software products along with product IDs and file information (such as when a file was posted, and the size and version of the file).

```
<software>
  <checkuptime interval="01:00:00"/>
  <product id="CBCC" uc="low" os="win32">
    <file name="cbcc_inst.exe" size="756345"
ver="3.00.00.00"
          posted="12/16/03 12:34 pm GMT"
md5="2d6897210ca5b816f24a7d52e319e07d"/>
    <desc>ChessBrain Control Center</desc>
    <changes>This release contains a fix
for...</changes>
  </product>
</software>
```

Arguably, the XML information could have been represented in a far more compact binary format. For example, the XML translates to a structure with precise data types, such as:

```
struct Software
{
   char product_id[20];
   char uc[4];
   char os[10];
   char filename[40];
   long filesize;
   char fileversion[12];
   char filemd5[33];
   char filedesc[80];
   char filechanges[1024];
};
```

Of immediate concern would be the use of the long filesize field, which would be interpreted differently across computers. The so-called "endian," which determines how computer memory is organized, causes computers to interpret the long filesize number in incompatible ways. This issue affects PCs and Macs, in addition to a host of other machines. Naturally, there are well-established methods of addressing this problem. The issue is that you must take those steps, and further place a similar burden on other programmers when they attempt to interpret the same data.

Another apparent limitation is the use of "fixed" data sizes. Misunderstandings and lack of proper data validation can lead to software defects, and one of the most common security threats in existence -- the dreaded buffer overflow. XML allows non-heterogeneous computers to safely receive and process text data.

XML's potential for enabling well-formed data makes it ideal for text parsing. Today, there are widely available programming libraries for working with XML using languages such as C/C++ and Java. In addition, all major scripting languages support XML, and remove much of the complexity of working with XML data. Because of its inherent structural simplicity, you're often able to quickly parse data using standard string runtime calls such as strcmp (), strstr(), and strncpy(), rather than employ the use of a full-fledged XML tool.

XML has proven to be a valuable tool and developers have created XMLRPC (XML for Remote Procedure Calls) and SOAP (the Simple Object Access Protocol) to formalize data exchange methods. SOAP has become the preferred method of building web-based services and now provides distributed computing applications with standardized methods of communicating with one another.

The use of HTTP, XML, and XML-based grammars helps to enable interoperability and dramatically improves the chances that your project will be scalable.

Server Design

The goal of a SuperNode server is to dispatch work units to PeerNode clients and to later collect results. There is no shortage of ways to accomplish those goals; however, because many distributed computing projects rely on HTTP as an underlying protocol, the use of a web server is by far the most common approach.

Web servers form the foundation on top of which a distributed application can be hosted. Using a scripting language such as ASP or PHP, a web server can be transformed into a custom application server.

Let's say that we've written a PeerNode client that processes a chunk of data and returns an XML-encoded result to our project's SuperNode server. Our SuperNode server is an Apache web server running the PHP scripting language with access to a local database server. We've written a script to process the incoming XML data and store the results in a database table.

The XML data might look like this:

```
<result>
  <client ID="phoenix34@yahoo.com" ver="3.12.02" />
  <workunit WUID="5570f980-96e0-44ab-92bb-
569ca73d591b" value=".0156892354334" />
</result>
```

The result package consists of a user's information (in this case, the user's email address), followed by the version number of the client software that was used to process the original data. Additionally the result package contains a work unit identifier, followed by a resulting value.

On the server side, we have a PHP script that looks something like this:

```
<?
   set_time_limit(300);
   header("Content-type: text/xml");

   function GetFieldData($string, $param)
   {
```

```
      $s = strstr($string, $param);
      if ($s)
      {
         $start = strpos($s,'"',0) + 1;
         $end = $start;
         while ($s[$end] != '"')
            $end++;
         $sval = substr($s,$start,$end - $start);
         return $sval;
      }
      return "";
   }

   $data = $HTTP_RAW_POST_DATA;
   :
   : More code here…
   :
?>
```

In the prior code fragment, the $data variable will contain the XML data sent from the PeerNode client. $HTTP_RAW_POST_DATA is a built-in PHP variable that holds the data that was received via an HTTP POST operation from a remote client.

Later in the script code, we'll use the GetFieldData() function to extract some of the XML values. For example:

```
$clientID = GetFieldData($data, "ID");
$workunitID = GetFieldData($data, "WUID");
$resultValue = GetFieldData($data, "value");
```

After extracting the values, we'll store them in a database using a code fragment like this:

```
$dbinsertstring = "INSERT INTO tblResults VALUES
($clientID, $workunitID, $resultValue)";
mysql_query($dbinsertstring);
```

In our PHP example, a single script was used to receive the XML data via HTTP, extract the data values, and finally, store the result in a database. While the specific function calls will be different when using other scripting languages, the general approach and ease of use remains similar.

Leveraging the proven flexibility, reliability, and robustness of a popular web server can be a wise decision. However, using a web server is entirely optional. After careful consideration, you may conclude that a custom server is the only viable approach for your project. The general operations would be roughly the same as our prior example, but taking a custom server approach would be considerably more complex. This is because developing a concurrent multi-threaded server application is a non-trivial task, and one worth avoiding if at all possible. However, if you're a glutton for punishment and enjoy TCP/IP programming, data structures, threads, and sleepless nights, a custom server might just be what the doctor ordered. Having written several custom servers, I can say that there are better ways of cultivating gray hairs. Fortunately for the intrepid developer, there are many great books available that explore server and protocol development.

Client Software Considerations

The development of the PeerNode client requires that you carefully consider several important issues; additionally, the complexity of the client software development will strongly depend on the approach you chose.

Let's consider a few vital questions:

- Which platforms will your project software support?
- Will you develop exclusively for Microsoft
- Windows, Linux, or Mac OSX?
- Will you attempt to support multiple platforms?
- Will your client software have external requirements, such as a dependency on a runtime library or framework such as Java or .NET?
- Will you build your client software using a high-level network programming library, or will you code using lower-level TCP/IP sockets.

Naturally, the more platforms your project supports, the broader your user base may become. This is a strong argument for supporting multiple platforms despite the resultant complexities. One way to sidestep many of the platform issues is to use a cross-platform development tool. For example, if you develop your client software using Java, then your users may only need the Java runtime software on their machines in order to run your client software. Another approach is to use a highly portable programming language such as C, C++, or Perl. Just keep in mind that while a core language may itself be portable, depending on runtime libraries and non-standardized language features can quickly compromise portability. A good example is developing a Windows client using the Microsoft Foundation Classes (MFC) and then trying to port the application for use on Linux. The key is to develop the application while using and testing on the target platforms. Don't introduce a feature unless you know for certain that the feature is present on all of your target platforms.

If you're not concerned with supporting multiple platforms, you might consider Microsoft .NET. At this time, .NET requires a 20MB framework download when used with older versions of Windows. Future releases of Windows will include a version of the .NET framework, so this may be less of an issue moving forward.

All of the programming languages we've considered in this article support the use of TCP/IP programming; however, the complexities increase depending upon the approach you take. For example, you can take a high-level approach, which shields you from specific network programming calls, or you can take the lower-level approach, where you'll handle the reading, writing, and buffering of network data.

The strongest argument for using a readymade framework, such as BOINC, is that many of the complexities of network programming are virtually eliminated.

Regardless of the software development approach you decide on, you'll still need to carefully consider usability issues. All software needs to be developed with an understanding of the target audience. Who will run the software? What skills are end users expected to posses? Understanding the answers to these and other important usability questions can directly translate to whether your project will be well received and ultimately, well-supported. To gain wide acceptance, your client software needs to shield end users from unnecessary complexities.

The SETI@home project was the first successful DC project to recognize the ubiquity of the PC screensaver as a vehicle for harnessing idle processing cycles. There are now several well-understood reasons why screensavers have proven ideal. Most people view screensavers as both innocent and non-intrusive. People know that screensavers only start when a machine is typically idle. Also, people in offices enjoy displaying their screensavers rather than simply turning off their monitors. This makes screensavers an ideal tool for advertising a project, often leading to a "word of mouth" marketing effect (See "The ChessBrain Project: A Global Effort To Build The World's Largest Chess SuperComputer," Justiniano, C and Frayn, (2003), Journal of the International Computer Games Association; ICGA Journal Vol. 26, No. 2, pp. 132-138.) Not all projects support the use of a screensaver; however, the merits are certainly worth considering.

Building your client software to require as little technical experience as humanly possible should be one of your most important goals.

Databases

The database server has become an indispensable fixture in distributed computing projects. While conventional wisdom dictates that databases are used for storing data, databases have matured into application-development tools that far exceed their originally intended uses.

Database servers are commonly distributed throughout a network and remotely accessed by other applications and servers. This allows distributed applications to communicate with one another by reading and writing database records. Is this the best way to perform inter-process communication? Perhaps not, but bear with me. Database servers also help to distribute an application's memory requirements among one or more servers. Without a database server, an application might need to maintain data structures in memory and on disk, taking away from the total available memory and negatively impacting overall performance. Database servers also help to distribute the data-processing load that results from the need to search, sort, and tabulate results. Finally, when properly utilized, database servers help glue together distributed applications to help achieve maximum scalability.

Application development can be a complex endeavor, and database servers have the potential of simplifying difficult tasks. There are powerful database systems available for every budget, leaving little reason not to use them in your own project.

Testing Considerations

Whether you choose to leverage existing servers and platforms (such as Apache, MySQL and PHP) or build your own custom server, it is vital that you consider how you'll test your project's software. An all-too-common mistake is to develop software without first considering how it will be tested. In the end, some software products cost more to test than they cost to actually develop in the first place.

Scripting languages provide a means of performing serious testing. Scripts can be used to test a server's functionality and its ability to withstand excessive stress. Scripts are easily modified and repurposed, and once written and tested, they allow for subsequent quality-assurance regression testing, ensuring that the server performs correctly after any recent modifications.

Another important consideration is to simulate end-user environments as closely as possible in order to perform accurate tests. It is essential to install proxy servers and firewalls during the software research and development stage to ensure proper behavior. In particular, the use of proxy caching servers should be carefully examined. A caching server seeks to reduce network traffic by storing and reusing data. This behavior can introduce problems for applications that use HTTP.

The use of open standards such as HTTP and XML ensure that both professional testing houses and informal testers can assist with the product testing. You see, interoperability does have its benefits.

Network Failures: Expecting the Unexpected

Building network software can seem deceptively simple at times, especially for less experienced developers. In network programming, the complex actions of humans, computers, and networks can cause unexpected behavior that results in a catastrophic outcome, such as a server crash or, worse, a system crash. The key to surviving catastrophes is to plan for them.

Consider these issues:

- The SuperNode server may send a task to a PeerNode client, which gladly accepts the task.

- Later, its user stops the client software before a result can be returned.
- A PeerNode client may be in the process of returning a result when it loses its network connection.
- The SuperNode server may collapse under the pressure of PeerNode connections, resulting in a crash, which leaves thousands (and possibly, millions) of PeerNodes unable to connect.

Unless carefully planned for, unexpected problems may ultimately destroy a distributed computing project that is ill-prepared to cope with unexpected situations.

It's essential to distribute the same work units to many PeerNodes. If one PeerNode does not return a result, perhaps some other PeerNode will. Should all of the PeerNodes working on the same task fail to return a result, then the work unit is simply marked as incomplete and will be sent to another batch of PeerNodes at a later time. The use of redundancy creates a robust system, where the project is not dependent on whether or not an individual PeerNode completes a task.

You must also take into account the potential failure of a SuperNode server. How will PeerNode clients respond if a SuperNode is no longer available? PeerNode clients must be designed to handle the case of unreachable SuperNodes. If a PeerNode client stopped working (because of a crash), then the moment a SuperNode server becomes unreachable, your entire network might fall apart.

One way to address this problem is to design your PeerNode so that it accepts connection failures and gracefully retries at a later time. Additionally, build your PeerNode client so that it's able to connect to Internet addresses by name, such as node01.distributedchess.net or node02.distributedchess.net. If a connection to node01.distributedchess.net fails, then the PeerNode will try node02. PeerNodes can also maintain a list of SuperNode servers and migrate to the next available server as needed. Should a SuperNode server fail, PeerNodes will behave like a swarm of bees changing direction on their way to another destination.

Security

Security plays a vital role in many aspects of a DC project. Both project organizers and participants have valid concerns regarding security. Participants have concerns about their privacy and their machine's susceptibility to viruses, and many wonder if using DC software makes their machines easier targets for attackers. On the other side of the fence, project developers have concerns that attackers may find ways to tamper with results, invalidating received work units.

Developers are faced with securing a project from a number of vantage points. A first order of business is to examine points of vulnerability. Where can an attacker cause harm? Which aspects of the project can be exploited and otherwise abused by participants?

Project participants download PeerNode client software from a project's web site, so it makes sense that the project's web server is an obvious target. If an attacker can penetrate a site and replace the downloadable PeerNode clients with compromised versions, then many machines will become infected.

Fortunately, a considerable amount of research has been done to address server security. Intrusion detection systems (IDSes) use sophisticated monitoring techniques to detect potential security issues. An IDS can monitor TCP packets to identify when an attacker is performing a port scan or when a denial-of-service attack is underway. IDSes can also monitor system files and user patterns for unexpected behavior (such as a normal user acquiring root level access), and when core system configuration files are modified. You can configure an IDS to send you an email when a problem occurs. Think of it as an early warning system.

There are thousands (OK, maybe just hundreds) of tools that can be used to monitor network traffic. One such tool is the freely available Ethereal, which uses a packet-sniffing library in order to perform its higher-level functions, such as filtering and display. The same types of underlying tools are available to attackers who can use them to intercept and modify data while in transit.

Take, for example, an attacker wishing to disrupt a DC project. The attacker builds a Trojan software product that masquerades as a useful monitoring and statistics tracking system for end users. The malicious tool performs useful functions while slightly modifying the transmitted results prior to sending them on their way. The tampered data has the potential of completely invalidating the DC project -- resulting in a complete waste of time for all involved. We won't get into the many psychological reasons why some people consider this sort of behavior exciting, but suffice it to say that disrupting a high-profile DC project might offer an attacker icon status in certain circles.

The only hope of protecting your project is to make it difficult for an attacker to modify transmitted data. As with most things, there are easy and harder ways of doing things.

Data Hiding

Software developers are sometimes faced with the classical problem of space versus performance. The need to protect data may be sufficiently clear; however, the cost of doing so may be prohibitive. Hiding data, rather then fully encrypting

it, and using strong validation techniques on both the server and client end, may offer a suitable compromise.

Data compression can effectively reduce bandwidth requirements, and has a positive side effect of masking the original contents. Applying byte transformations, such as XOR operations and weak reversible data ciphers, will further aid in data hiding. Clearly, data hiding is by no means as secure as data encryption, but may be suitable for use in certain settings.

Data Encryption

Widely available implementations of popular data-encryption algorithms leave project developers with little reason not to apply some form of data protection. One popular algorithm is the Advanced Encryption Standard (AES), also known as Rijndael (pronounced "rain doll"). AES is a variable block symmetric encryption algorithm developed by Belgian cryptographers Joan Daemen and Vincent Rijmen as a replacement for the aging DES and Triple DES standards that are still commonly used to secure e-commerce. AES is currently used in hundreds of high-end encryption products and is a favorite among developers. Additionally, AES implementations can be found online.

For maximum security, where performance may be less of an issue, the use of public key cryptography is highly recommended. Public Key Infrastructure (PKI) systems use public key encryption to create digital certificates, which are managed by certificate authorities. Certificate Authorities (CAs) establish a trust hierarchy, which can be used to validate authenticity through association. The use of PKI would allow a SuperNode server to authenticate PeerNodes, and PeerNodes to validate that they are indeed communicating with an authentic SuperNode.

Detecting Software Tampering

Project contributors may acquire PeerNode client software from one of many locations. For example, a project team leader might download and place the client software on the team's web site along with specialized instructions for team members.

There is a certain degree of trust associated when the software is downloaded directly from the project's main web site. However, when project software is made available from different locations, project contributors may not be able to trust the origins and validity of the software. For all they know, the software could be a Trojan program. To address this concern, DC software is often posted on a project's site along with a cryptographic hash string, such as:

```
3402b30a24dc4d248c7c207e9632479a  client21201-01-
lin-i586.tgz
```

The string of numbers and letters is the output of a program called md5sum, which generates the string of alphanumeric characters when given a filename as input.

End users can download a project's client software and type:

```
md5sum client21201-01-lin-i586.tgz
```

The output is a string that should match the one posted on the download site.

For higher levels of security, some projects sign their files using a private key. Users wanting to validate that a file's digital signature is correct can retrieve the project's public key (available online via public key servers) and use it to validate that the downloaded file. The signature below is an example of what a contributor might see posted on the project's site.

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.2.1 (GNU/Linux)

iD8DBQA/ye8su9d1K+MjI6sRAtQXAKCgcXahYj1ZcptsXR10WCn
SbKs2ggCeK/Qv
4THuyfGOeDEyHiHnHX9pkZw=
=Nj+a
-----END PGP SIGNATURE-----
```

The important thing concerning a cryptographic hash and a digital signature is that both techniques can be used to determine whether a file has been tampered with after it was posted. This allows the software to be distributed and for end users to validate that the file has not been tampered with. By far the most secure method involves the use of digital keys, and that technique is being used in an increasing number of projects.

There is a wealth of security information available on the Internet, and many open source projects demonstrate working implementations. As a DC project developer, you have the responsibility to explore security and protect both your project and your members.

Combating Aging: Software Updates

As computer users, we've grown accustomed to automatic software updates. Now companies such as Microsoft, Apple, and Red Hat offer their customers software updates. They're not alone, as thousands of other software companies also offer updates.

Updates are released for any number of valid reasons, ranging from program fixes and new features, to the latest security Band-Aids. In all fairness, software complexity has increased while the time to market has decreased, causing products to be released to an unsuspecting public in a less-than-perfect state. In addition, network-connected software is under siege as attackers attempt to discover and later exploit security flaws. Rapidly distributing software updates has

become the only real defense companies have against coping with the unexpected.

A real issue facing developers is how to make software updates quick and painless for their customers. Long gone are the days when posting an update on your company's FTP or web site sufficed. Companies are seeing their products used by a wider demographic, where even a "one-click install" is one click too many.

The challenge affects all DC projects, to lesser and greater degrees. Long-running, static-type projects often do not require software updates. However, for projects that are highly dynamic, where the client code is being updated in response to ongoing research and development, the need to release continuous updates is far more critical.

Bandwidth and Hosting

An important consideration for any DC project involves determining network bandwidth utilization. It is important to consider bandwidth from both the SuperNode server's end as well as from a PeerNode's perspective.

Many popular DC projects package data into chunks that take PeerNodes days or sometimes weeks to process. In these cases, the actual network bandwidth utilization on the client side is negligible by average end-user network usage patterns. For example, accessing a single high-bandwidth web site with lots of graphics can use more bandwidth than most DC clients use in weeks.

The situation on the server side offers a very different perspective, where a single SuperNode server might service thousands (or, if truly popular, millions) of requests per day.

It is important to examine the frequency and size of data transmissions and to adequately plan for bandwidth requirements. One of the best ways to study the behavior of a network application is to use a network analyzer. Ethereal is a good tool for examining what is actually going through the wire; however, network-bandwidth measuring tools and load-test simulations should also be used to gain better insights into the application's true requirements.

Understanding the project's network requirements is necessary in order to choose the right server-hosting plan. Proper planning is essential, because if your project becomes successful, you may discover you can not afford to pay for the bandwidth costs. Even if you do not pay for bandwidth, the bottom line is that someone, somewhere, does pay, and without proper analysis, your project may be terminated prematurely.

The best course of action is to plan for bandwidth, carefully consider your data protocol, and potentially, use data-compression techniques.

Backup

Your favorite computer's days are numbered: it's just a matter of time before a key component, such as a hard drive or power supply, breaks down. Because distributed computation projects typically deal with vast amounts of data, it is absolutely vital that you develop a backup strategy.

It is virtually impossible (or exceedingly expensive) to guard against data loss. The key is to minimize your risk exposure. For instance, if you perform a backup once per day, it's possible that you may lose nearly a day's worth of data in the event of a hardware failure. Thus, backing up data once per hour minimizes the risk of losing an entire day's work. Data mirroring using redundant hardware is certainly the way to go if you can afford it, however you'll still need a backup policy.

An overwhelming majority of individuals don't perform regular data backups. The reason is actually quite simple -- it's a chore to do so. The best way to ensure that data is backed up regularly is to automate the process. On UNIX-type systems, the process is greatly simplified using the crontab scheduling system and archive scripts. On other systems, you may need to explore backup software solutions.

Another key backup strategy is the concept of offsite storage. In addition to redundant storage (storage on multiple machines) and CD archives, I use a service called Xdrive as an offsite storage facility.

Protect your data. Hardware may be expendable, but loss of data may cripple your project. Yes, this is potentially one place where paranoia may really pay off.

A Value Proposition

We've skimmed the surface of many, but by no means all, of the technical considerations you might encounter. However, not all of the issues you'll encounter will be technical in nature. There is a very human aspect to distributed computing, and failure to understand the human elements will seriously jeopardize your project's long term viability.

In the past, the notion that individuals would pay for, and allow their computers to participate in, research projects was foolhardy, at best. Times have changed. Today, millions of people participate in distributed computation projects. As a result, we are now able to tap a wealth of computing resources. However, there is one small catch: we must convince people to join our projects.

If you are interested in getting people to join your project, you need to create a value proposition offering an enjoyable and rewarding experience in exchange for participation. In addition, you need to consider how to retain members once

they have joined. The best way to begin to address these issues is by understanding the underlying motivators that attract people to distributed computation projects.

Why People Join DC Projects

You may be wondering what drives a person to contribute their time, energy, and the use of their computer to a distributed computation project. Although specific reasons vary, there are a few common themes that consistently appear in DC projects.

A Sense of Purpose

Some members are motivated by a deep sense of purpose. Projects such as FightAids@home and the University of Oxford's cancer research project offer individuals the opportunity to support noble research that might ultimately benefit millions of people.

A Sense of Community

Many active members enjoy being part of a community and collaborating with other people. Generally, people like to be involved in things that transcend them as individuals.

Competitive Opportunities and Peer Recognition

Members want to know that their contributions matter. All major distributed computing projects track member contributions and post the results on the project's web site. Members gain the respect of their peers and obtain subculture ranking within communities.

Entertainment

Participating in a distributed computing project can be entertaining in a number of ways. Meeting people and competing against them can be entertaining.

Successful projects understand the needs we've just examined and seek ways of promoting them within the context of the project.

The Distributed Computing Scene

Distributed computing projects have given birth to communities of enthusiasts who closely support projects. In turn, project web sites publicly display project statistics and member ranking (sometimes referred to as leaderboards) offering individuals a convenient way to compare their ranking against those of their peers. This has led community members to form teams, which compete against one another to see which group can make the most significant contributions to a project. Project organizers are eager to support competition because the results typically lead to teams recruiting more members and subsequently, more computers.

Distributed computing team members have adopted the moniker "DC Team," and members refer to themselves as "DC'ers." Many DC'ers take their hobby seriously, and many run two or more machines, with some running as many as 40 or more while participating in various projects.

When asked why he contributes to projects, DC'er Chris Harrell replied, "I like to think I solely pursue DC projects for the common good of mankind, but I cannot deny the fact that the project statistics are the main attraction for 99% of DC contributors." Chris is far from alone; for many DC'ers, interest in a project comes second to competing for public ranking.

When I started ChessBrain, a global project to build the world's largest distributed chess computer, I was surprised to discover contributors who had very little interest in the game of chess. This was my first introduction to a network economy where DC teams support research projects in exchange for an opportunity to compete against one another.

International teams, like the Dutch Power Cows and AnandTech, claim to have thousands of members. DC Teams have become a powerful force in helping to shape the future of distributed computation projects on the net, by providing a highly technical member base with access to thousands of machines. They are the unsung heroes of a new age.

Establishing Credibility and Trust

Project participants have many projects to choose from, but don't mind exploring a project for a brief time, in order to get a sense of it. However, potential members won't waste their time participating in a project that doesn't appear worthwhile.

Before a significant number of people take interest in a new project, it must first establish a certain degree of credibility. Establishing credibility begins by clearly articulating goals and demonstrating the project's commitment to achieving a measurable result. The project must clearly communicate the message: "This project is worth your time!"

Most distributed computing projects maintain web sites, which articulate the project goals, present project status reports, and offer software download areas. In some cases, project web sites feature online forums where members can post feedback directly to the development team and other members. A project's community forum offers project leaders and members opportunities to publicly engage in conversations. The presence of a public forum can go a long way toward communicating the commitment and seriousness of a project.

One way of gaining credibility is through association. Some DC projects enjoy near-instant credibility when well-established institutions or well-known companies sponsor them.

In the process of building credibility, you must also establish a relationship based on trust. Generally, participants must believe in the credibility and trustworthiness of a project before downloading and running potentially malicious software on their machines and networks.

One of the surest ways of establishing trust is to engage in direct conversations with potential members via email, on a project web site, and on other public forums. Nothing says, "your voice matters" faster than a prompt reply to a member's inquiry. Although this isn't always possible, the goodwill generated is worth its weight in gold.

Open and honest communication is a tool that tears down relationship barriers, and helps foster healthy and productive relationships. This is a point that is often difficult to remember when coping with difficult people. Let's face it – public relations can be a difficult job. Freedom of networked speech often results in members pretty much saying whatever they want while publicly venting frustrations. These sorts of behavior can quickly erode a project's credibility as mob-like conditions lead others to join in. This is where, as a leader, you must exercise the most restraint. Months and possibly years of relationship building can quickly crumble as a result of an ill-prepared response.

It is important to remember that project contributors give freely of themselves and that it is difficult to run a distributed computation project without them. Exercising tempered restraint and maintaining an eternal state of gratitude is vital to maintaining a successful project.

The most important element in a distributed computation project remains the people and communities who join together to unlock the vast potential of distributed machines. To paraphrase the Matrix: They are the gate keepers. They are guarding all the doors. They are holding all the keys.

Years ago, Sun Microsystems promoted their marketing slogan: "The network is the computer." Although this still remains relevant in the context of distributed computing, I'd like to offer another mantra: "The people are the network." The machines are simply tools that allow us to touch, if for just a moment, the very limits of our imaginations.