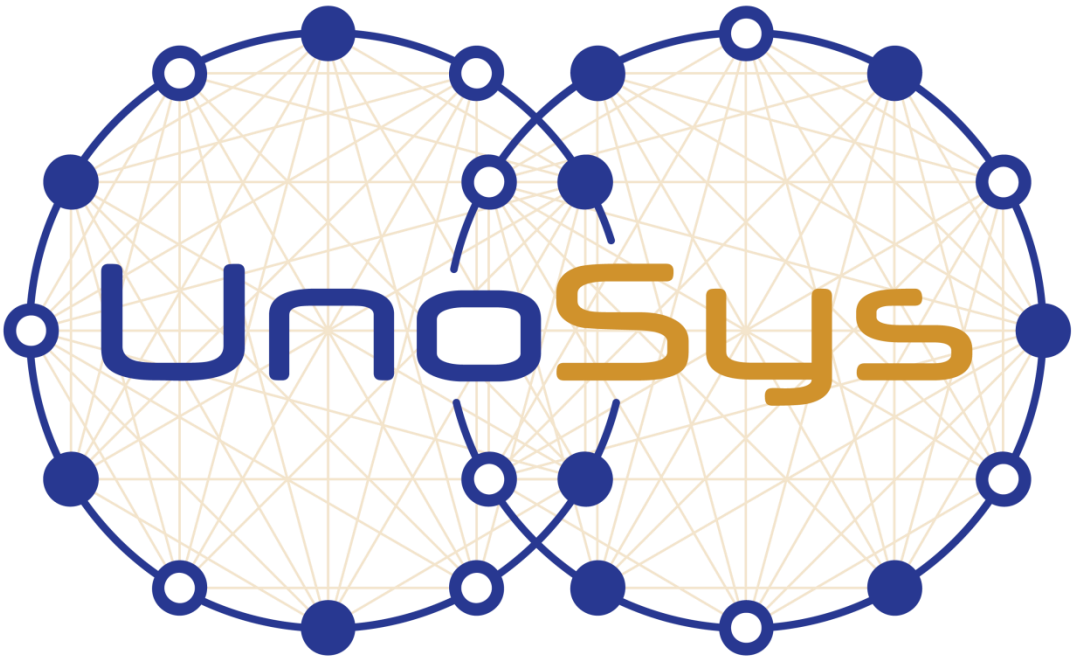
UnoSys – An Overview of the Operating System Design for the World Computer



The World Computer Organization

www.WorldComputer.org

July 2020

Table of Contents

1.0 Introduction to UnoSys 3
1.1 What is UnoSys? 3
1.2 Internet Scale P2P 3
1.3 Distributed System Reliability 4
1.3.1 Safety and Liveliness 4
1.3.2 Fault Tolerance 5
1.3.3 Trade Offs 7
1.3.4 Byzantine Fault Tolerance 8
1.4 The UnoSys Processor 9
1.4.1 UnoSys Applications 9
1.4.2 UnoSys I/O Services 10
1.5 Fully Distributed Under the Hood 11
1.5.1 Reassembling Blocks 11
1.6 Unique Security Challenges of Permissionless P2P Distributed Systems 12
1.6.1 Untrusted System Administrators 13
1.7 UnoSys – A Fortress of Security 14
1.7.1 Securing End-User Data on Peers 14
1.7.2 Securing Data Passed Between Peers 17
1.7.3 Securing UnoSys Processor Code 18
1.7.4 Byzantine Fault Tolerance Revisited 25
1.7.5 Cryptographic Key Management 29
1.7.6 Self-Organization 30
1.8 UnoSys Technology Summary 32

1.0 Introduction to UnoSys

Every non-trivial computer has an operating system to provide the services that developers need to write applications for it, and the World Computer is no different. This document provides a detailed overview of UnoSys, the operating system for the World Computer Project. It covers what UnoSys is, the challenges it was designed to solve, and the technology used to meet those challenges.

1.1 What is UnoSys?

At its core, ***UnoSys is a modern, reliable distributed peer-to-peer (P2P) operating system for the Internet.*** The purpose of this operating system is to provide a *single computer abstraction* for developers to write a new generation of Internet scale applications on. UnoSys gets its name from the Italian word "Uno" meaning *one* and "Sys" for *system*. UnoSys exposes an easy to understand and reason about *one system* abstraction to applications written for it. This permeates its design decisions and is central to the simplicity of the approach UnoSys takes to achieving massively scalable distributed computing.

Applications built for the simplified single computer abstraction UnoSys exposes become seamlessly scalable across a vast distributed network merely by running them. This one elegant global computer model is designed to support *any* kind of application ranging from regular corporate business (i.e., database) applications, to consumer media applications, including on-demand media players and real-time chat and video collaboration applications, including games, all the way up to cryptocurrency distributed ledgers and big data analytics applications that deal with Petabyte and beyond data sources. Moreover, these applications – from the trivial contact list, to the most sophisticated CRM or ERP system – can be written in the developer's language of choice and then run on the fully distributed UnoSys operating system, seamlessly providing them reliability at Internet scale, regardless of size or data requirement.

1.2 Internet Scale P2P

As with any *Peer-to-Peer* (*P2P*) system, UnoSys is designed to run on a network of computers. Each unique computer in the network is known as a *peer*, and the network can be of any size. Although it is fully expected that UnoSys will be of interest to enterprises looking to create their own *private* UnoSys single-computer *instances* running collectively on servers confined within their internal corporate networks, it must be stressed that the most compelling use case for UnoSys is in the creation of a single *public* world-computer abstraction running on the open Internet. Achieving this is *the* major design goal of UnoSys from the onset.

While UnoSys *runs* on a network of <u>many</u> peers it only *exposes* a <u>single</u> computer abstraction. At its core, this means that applications (and therefore the developers that write those applications and the users that run those applications) need never concern themselves with the logical notion of a *network*. P2P infers peers within the UnoSys operating system have the ability to (and do indeed) communicate with each other to achieve coordination. However, this is completely abstracted away from the applications and their users. UnoSys, as the distributed operating system, encapsulates the

responsibility of all required coordination through communication between its peers so that the application developers and users can remain blissfully unware of anything but a single local computer abstraction. It turns out that removing the notion of a network radically simplifies everything to do with software development. Sun Microsystems almost had it right all those years ago when they famously said "*the network is the computer*". Instead, with UnoSys "*the computer is the network*".

The UnoSys nodes (peers) all run the same software. They are indeed *peers* in that they have identical capabilities. Peers can both send and receive messages to each other. This means they act as both clients and servers with respect to the coordination of the events that transpire between them. It is in this context that we say that all of the nodes collectively make up *the* (single) UnoSys (operating system) computer abstraction.

For people outside of the networking world, P2P is often associated with "bad software". Napster (1999) was perhaps the most infamous example of a wildly successful consumer-oriented P2P application, which reportedly enabled at its height over 80 million people to share music files with each other, largely illegally due to copyright infringement. Authorities were able to eventually shutdown Napster because a single core aspect of its architecture was in fact centralized. There is of course nothing bad or illegal about P2P technology itself, only the way it has been (mis)used in the past. Indeed there is no better way to scale a distributed system than via P2P networks. In fact, the Internet itself is defined by P2P communications at its very foundation.

UnoSys utilizes proven P2P technology to build a fully decentralized *network overlay* across all peers in order that they can *globally* self-organize and coordinate using nothing more than pair-wise interactions. By abstracting the complexity of all coordination and communication away into an operating system layer of software common to all peers, seamless application distributivity along with significant application development simplification can be achieved.

1.3 Distributed System Reliability

As with any operating system, UnoSys must be reliable to be useful as a platform to develop applications on. Building reliable distributed systems is very difficult, especially in *open public* networks such as the Internet. ***UnoSys seeks to answer the question of how to build an open, public, reliable single programmable computer abstraction from a network of inherently unreliable and possibly even malicious computers.*** The answer to this question, in a word, is *carefully*.

1.3.1 Safety and Liveliness

The starting point for ensuring *reliability* in a distributed operating system is to address the two key academic concepts of *safety* and *liveliness*. *Safety* is the idea that the system is operating *correctly* (i.e., is behaving as intended). It is the intuitive notion that the system's overall integrity is *good* – i.e., it remains safe (enough) from harm to continue performing its intended function correctly, notwithstanding any attack it might be under from a hostile agent. In contrast, *Liveliness*, is the notion that the system is available to make progress (i.e., continue providing its core function) even in the face of multiple nodes failing. So in good conditions where all nodes of a distributed system are running

correctly and responsively and the network is fully connected, the system would have safety and liveliness[1].

However, in any network, especially the network of networks known as the Internet, computers *fail*[2] all the time for any number of reasons. For example the computers themselves can crash or laptop covers can be closed, or the network gateways (e.g., the Internet Service Provider) they use could be down, all of which make the computers unreachable. Worse yet, a node can fail in so-called *Byzantine* (i.e., arbitrary) ways where a malicious agent like a bot or virus is purposely delaying (perhaps indefinitely), duplicating and/or deleting the messages the node sends to other nodes in an attempt to cause harm to the system. An even more sinister scenario is a nefarious agent taking control of, and even altering, the actual *code* a node is running, causing it to appear to be functioning correctly, but in fact forcing it to arbitrarily misbehave by counterfeiting messages that were never sent (i.e., create them out of thin air), or deliberately returning the wrong responses to request messages.

Correctness in a distributed system is achieved by faithfully implementing in code distributed algorithms that have been mathematically proven to be correct. While the algorithms are often very complex, they have been derived and vetted by years of careful peer-review. Therefore, implementation can be achieved with methodical software engineering practices for the most part, and the resulting software can be determined to be correct through careful verification of adherence to the proven published algorithm. However, correctness of a distributed algorithm generally requires pair-wise coordination between specific nodes within the system. Thus node failure, in all its forms, undermines system correctness once the distributed algorithm's specified fault tolerance threshold is exceeded. This then impacts the overall safety and/or liveliness of the system.

The core challenge faced by every distributed system is to protect the correctness of the overall system through safety properties, while simultaneously ensuring high availability of the system through liveliness properties. Without both safety and liveliness, the system simply cannot achieve correctness according to its purpose and therefore is not reliable, rendering it practically useless from a utility perspective.

1.3.2 Fault Tolerance

In pragmatic terms, the ability to *tolerate network partitioning* (i.e., some percentage of peers in a P2P network failing) by continuing to operate correctly – i.e., safely with liveliness – defines the essential challenge in designing distributed systems. And no meaningful conversation about fault tolerance can take place without the concept of *data replication* quickly entering the discussion. Since a system cannot continue to run correctly if it requires data that resides on a failed node, *data replication* quickly

[1] Assuming that the distributed algorithms being used have been coded properly.
[2] In this context *fail* simple means they are unavailable

establishes itself as a necessary condition to achieving fault tolerance in distributed systems. Indeed, it is largely the data that is *distributed* (i.e., replicated) in fault tolerant distributed systems[3].

Algorithmically speaking, distributed systems replicate data values to multiple nodes to provide fault tolerance. Should one node crash a data value can still be accessed on other nodes storing replicas in order to progress. In this manner, the failed node has been *tolerated* without compromising system reliability – i.e., its safety or liveliness. However, *pragmatically* speaking, data replication across nodes can also provide two other performance benefits. First, replication can potentially bring data *closer* to the client accessing it within the network thereby reducing data access *latency*. Second, replication potentially spreads the load in resolving data requests across the replicated nodes thereby improving data access *throughput*. Taken together these potential performance benefits are how *scale* is achieved in distributed systems.

Importantly, however, algorithmic correctness does not concern itself with scale. Performance gains through scalability are pragmatic implementation details only and are typically of no consequence to the mathematical proof of correctness of a distributed algorithm. Nonetheless, it is often these same pragmatic real-world implementation details that make the difference between theoretically interesting yet completely impractical distributed system designs, and those that stand a chance of actually working in real-world constrained environments at acceptable performance levels.

The often overlooked danger in augmenting complex *correct* distributed algorithms with real-world performance optimizations is that it is very easy to break their correctness in very subtle ways. If this happens, the result can (i.e., will eventually) lead to very difficult to find non-deterministic (i.e., rare and difficult to reproduce) bugs that compromise overall system integrity. Moreover, heavily optimized distributed algorithms often can no longer rely on their original mathematical proofs of correctness, and need to have those proofs re-established to ensure the system's safe design – a decidedly non-trivial exercise. This underscores the need for software engineers to have a very strong understanding of the base distributed algorithms they are implementing before attempting to optimize them to meet real-world constraints. Nevertheless, being scalable is paramount in achieving robust distributed systems and so distributed algorithms *will* be optimized so that data can be replicated efficiently.

The benefits of data replication are easily understood in the context of read-only data. However, things complicate quickly when data value *updates* are permitted since replicas now need to be kept synchronized with each other. Failure to do so compromises system safety because the *single system view* of the data value becomes divergent across the distributed system (i.e., different clients see different values for a data value at the same moment in time depending on which replica they read it from). Therefore, the more times a data value is replicated, the greater the effort involved in updating that data value so that it remains consistent across the distributed system.

[3] In some distributed systems it is the code and not the data that is distributed, but these are more traditionally known as parallel computing systems rather than distributed systems. We use the term distributed systems to more specifically refer to data replication systems. Parallel computing systems can be thought of as a special case of distributed systems where the *data* being distributed is the code (i.e., binaries) itself.

The next obvious question then is how much data replication is the right amount? Intuitively, increasing the number of replication nodes for a data value increases the resiliency of the overall distributed system because a greater number of node failures can be tolerated while still being able to serve data access requests – i.e., remain lively. Furthermore, increasing replication nodes also provides better opportunities to scale, albeit at the cost of more overhead in keeping those replicas in sync. Conversely, reducing the number of replication nodes improves performance during updates because fewer replicas have to be kept in sync. But this comes at the cost of lowering fault tolerance since the system as a whole cannot tolerate as many failed nodes before impacting its liveliness. In this way, reducing replication is the antithesis of scale.

1.3.3 Trade Offs

This iconic tug-of-ware between tradeoffs in liveliness and safety in the face of faults within distributed systems design is classically captured in the so-called *CAP Theorem* which states that of the three fundamental properties we would like our distributed systems to provide; data *Consistency*, data *Availability* and network *Partition Tolerance*, you are allowed, in fact, to choose only two. That is, you can design your system to maximize *Consistency* and *Availability* (CA) or *Availability* and *Partition* Tolerance (AP) or *Consistency* and *Partition* Tolerance (CP), but you can never devise a distributed system that exhibits all three of these properties at the same time.

We can simplify the choice here when building *real-world* distributed systems such as UnoSys by observing that for anything but purely theoretically designs, Partition Tolerance is an absolute requirement since you cannot achieve overall system reliability guarantees without it. Therefore, the CA systems model option from above can be discarded leaving only two possible diametrically opposed choices for real-world distributed system designs:

I. AP – Available and Partition Tolerant
II. CP – Consistent and Partition Tolerant.

In general, AP distributed systems provide better performance at the tradeoff of sacrificing some safety by occasionally serving inconsistent (e.g., stale) data. Whereas, CP systems guarantee *consistent* (e.g., safe single view state) data is always served, at the expense of some liveliness through added (sometimes significant) performance overhead. It turns out these options are opposite ends of a spectrum of availability vs. consistency tradeoffs in the context of partition tolerance, and that all positions on this continuum have their place in a distributed system depending on its intended purpose.

However, in practice, in order to reason deterministically about the behavior of a distributed system, application developers need to fully understand the exact availability vs. consistency guarantees it offers, which forces the distributed system implementer to declare where on the spectrum their system operates. For some purposes, occasionally not getting the most current data is perfectly acceptable if it increases performance (e.g., obtaining the current top score in a game), while for other purposes, single view state consistency of the data is mandatory regardless of performance implications (e.g., obtaining the current balance in a cryptocurrency wallet). UnoSys supports both extremes of the spectrum by

offering the developer a choice between data storage guarantees that are lively but less safe or safe but less lively.

1.3.4 Byzantine Fault Tolerance

While it is important for a distributed system to be explicit about its safety and liveliness properties, they must be understood within an explicitly defined threshold of fault tolerance. Fault tolerance is the single most influential characteristic of a distributed system as it is the stick by which the safety and liveliness properties of the system are measured. It is vital for the system to be explicit about identifying the level of fault tolerance it promises (as there are in fact many levels), so that precise reliability guarantees can be made about the system. In the harsh reality of today's increasingly insecure digital world, a viable distributed system targeting the open Internet must proactively prepare for the most hostile environment imaginable. Therefore, it effectively must provide the strongest fault tolerance guarantee possible – known as *Byzantine Fault Tolerant* (BFT).

BFT is the gold standard in fault tolerance. A system that makes BFT guarantees is claiming it is resilient to all manner of node *failure* – whether connection based or the result of malicious takeover. An even stronger version of BFT is known as *asynchronous-BFT* which is Byzantine fault tolerance in *asynchronous networks* – networks that do not offer ordered message delivery communication assurances (e.g., User Datagram Protocol (UDP) over IP). Asynchronous networks complicate matters because they make node *crash detection* impossible as there is no discernable difference between a node delayed arbitrarily long and one crashing outright. Further still, work on the famous Byzantine General's Problem led to a famous theorem that states that a network cannot guarantee consensus on safety (data consistency) if more than one third of data replica nodes in a partially synchronous network fail. This result is a "law of physics" with respect to distributed systems and it must be compensated for by making sure we store enough replicas to support a liveliness guarantee within a configurable node failure threshold. But as we have seen, more replicas create more performance incurring overhead to keep them all synchronized during updates within a spectrum of many possible safety guarantees.

Achieving asynchronous-BFT (aBFT) is the Holy Grail of distributed systems and is very difficult – yet essential for a distributed system operating publically on the Internet. An a-BFT distributed system must deal with any combination of *failure* due to a wide range of reasons:

1. Nodes crash
2. Nodes are unreachable (i.e. stop responding)
3. Nodes are reachable but Messages are delayed (possibly indefinitely)
4. Messages arrive out of order
5. Messages are dropped (i.e., never sent)
6. Messages are sent multiple times
7. Messages are invented out of thin air
8. Messages are eavesdropped on but otherwise not altered
9. Corrupt Messages are sent
10. Legal but wrong Messages are sent

When considered altogether, these challenges appear overwhelming. However, UnoSys addresses them all by breaking them down and tackling them independently, to achieve a-BFT with clear reliability guarantees for the overall operating system, where there is no such guarantee in general from the individual nodes that it comprises. However, before we discuss how UnoSys achieves a-BFT, we need to take a closer look at how the operating system works and how it achieves the extreme security it requires.

1.4 The UnoSys Processor

A tenet of elegant software design is "few, small and simple interfaces". UnoSys exemplifies this tenet by exposing only a small yet powerful set of primitives to application developers with which to do distributed stream-based I/O operations. UnoSys provides a base set of services you would expect from any operating system, as well as a few state-of-the-art services that will surprise and delight you. These exposed services are encapsulated into the UnoSys operating system *software* known as the *UnoSys Processor* executing as a single, separate, physical process on each peer node. The services exposed by the UnoSys Processor make up the publically accessible UnoSys *Application Programmer Interface* (API), which is easily accessed by any developer from any modern development language, to build any conceivable application. Applications communicate locally (only) to the UnoSys Processor running on the same computer (i.e., inter-process communication) via the API, and the Processor in turn communicates to other UnoSys Processors, as required, to provide the requested service. By using the simple local computer abstraction exposed by the UnoSys API for all application data I/O needs, a developer can easily create next-generation, seamlessly distributed, Internet-scale applications.

1.4.1 UnoSys Applications

The single computer abstraction model of UnoSys significantly simplifies how a developer writes applications to take advantage of the platform. Essentially, a developer can use familiar, *local* programming abstractions such as file streams or database tables, to write fully distributed applications that scale massively. Like other modern operating systems, UnoSys does not dictate what development language or tool set is used to develop applications for it. A developer can choose any language to write her applications in, as long as it has a standard HTTP communications stack. For example, applications can be written in C, C++, C#, Java, Python, Go, etc., compile to an executable or VM, or run interpretively as a scripting language such as JavaScript, as per the developer's preference or need. Furthermore, developers are free to control the look and feel of their applications, choosing their user interface experience as they see fit. Hence, as a platform, UnoSys is open to the widest possible set of applications and developers.

Strictly speaking, the developer's application does not run *on* the UnoSys platform. Rather it executes as its own separate process alongside the UnoSys Processor on the host node. The application can use the UnoSys API to call into the UnoSys Processor to consume the advanced fully distributed services it exposes for its distributed I/O needs[4]. *Mixed mode* applications can be built that use UnoSys API calls

[4] The UnoSys HTTP-based API means that applications can consume UnoSys services by making standard REST based web service calls. Importantly, the applications are permitted to only talk to the local UnoSys Processor.

when scalable distributed files and database tables are required and regular native files and database tables for all other more traditional I/O. The choice of when to use UnoSys distributed services is entirely up to the developer, allowing UnoSys to be adopted as much or as little as necessary to meet the application need. The only requirement to using UnoSys is that the application be run on a node running the UnoSys Processor process. To make a UnoSys application widely available to other UnoSys peers, a developer need only copy the application files to a publically accessible file system folder of the single UnoSys computer abstraction that all peers see.

1.4.2 UnoSys I/O Services

As you would expect, UnoSys contains a modern, efficient, secure *File System*. UnoSys exposes familiar *Volume*, *Directory* and *File object* abstractions enabling fast, secure, reliable file stream-based I/O against an abstract persistent store volume. Developers create a new object, or open an existing object, in the usual way by providing a path to it. All objects appear local to the user of UnoSys applications regardless of the peer they run on. There is no concept of a file share because there is no network. Objects can be created, opened, read, written and deleted securely from applications running on *any* UnoSys Processor using its one and only unique *local* path. In this way, these objects are globally accessible to all users, subject to the granting of appropriate security permissions decided by their *owner*.

UnoSys also contains base services you might not expect from an operating system. For example, it contains a built-in *Database System*. Similar to the file system, any application can create a new or open an existing database (a collection of tables and index objects) using its path and begin reading and writing data from/to the tables from any node in the system, again subject to security permissions granted by the owner. There is no centralized database *server* because there is no network. All access to the database is provided using its local path. As a distributed database system, UnoSys is much more than a primitive distributed key-value store. Tables have versionable schemas supporting rich nullable data types that can be accessed both sequentially and randomly based on dynamically selectable active index orders.

Due to the single computer abstraction model of UnoSys, all file and table resources accessed by applications appear logically local to the Processor node the application is running on. There is just one logical UnoSys *Computer* and all files and databases live on it. Applications are therefore single computer in context since they can access all required files or database tables from the one computer *logically* locally regardless of which Processor it runs on physically. This is critical to realizing the elegance of the model. Applications running in the UnoSys computing model no longer have to communicate directly with other computers to gain access to required files and database resources that live on them as they must in a client-server computing model. These resources are now simply and effectively “shared” between all applications running on the UnoSys operating system, by virtue of the

---

That is, all API URLs will reference *localhost* only. Specifically, it is not possible for a developer’s application to directly call a UnoSys Processor on another computer, as this would represent a networked client-server architecture that the UnoSys computing model works tirelessly to avoid.

fact that every application sees them as local resources, subject to granted security permissions. Together, the built-in UnoSys File and Database systems provide rich stream-based I/O services in a simplified logically local single compute model. However, the reality of where that actual data is stored is decidedly *not* local, but rather deliberately and massively distributed.

1.5 Fully Distributed Under the Hood

Even though UnoSys provides a logically local single computer abstraction, that abstraction is only *logically* local. There is in fact little that is *physically* local about UnoSys under the hood. In UnoSys, collections of Processors work in unison to create a single massive storage abstraction in the familiar form of a hard drive file system volume or database. To accomplish this, the atomic datum of those files and databases is stored across *all* of the nodes *uniformly*. Files are broken into equal size blocks where the block size is configurable for each file at creation time (the default file block size is 64K). Likewise, database tables are partitioned into blocks constituting a unique *record*, and indexes are partitioned into blocks constituting a unique *page*. Each of these blocks is given an immutable unique identifier which stays with it from its birth (creation) to its death (deletion). UnoSys ensures that given one of these unique identifiers, any Processor can *always* retrieve the block in a single pair-wise communication with the Processor that stores it.

These individual data blocks are distributed across all UnoSys Processors in such a way that arbitrary blocks are stored on arbitrary UnoSys Processors. Moreover, each block is replicated to multiple Processors according to a configurable fault tolerance threshold configured on a per file or database basis at creation time, in order to achieve high availability. Ensuring uniformity of the distribution of these blocks across all participating Processors is paramount to evenly sharing the load on the UnoSys operating system incurred by serving block read requests. By ensuring the blocks are evenly spread out across the nodes of the P2P system, overloading particular servers can be avoided since all nodes participate in serving the blocks. Indeed, uniformity of block distribution is paramount to the design of UnoSys and great care is taken to achieve it.

1.5.1 Reassembling Blocks

It is relatively easy to come up with a scheme to compress 1GB of data into 7 bytes. However, decompression without loss of data is the tricky part. Likewise, partitioning files and database tables and indexes into arbitrary blocks and spreading them across a large number of nodes in a P2P distributed system is not particularly hard. It turns out that bringing all those blocks back together on-demand to re-realize the original file, table and index abstractions is the hard part. Fortunately UnoSys has been carefully designed to do just that very efficiently.

The file system built into UnoSys is a state-of-the-art, parallelized and distributed, block-oriented file system. For example, when an application using the UnoSys API creates a file stream and writes 1MB of data, under the hood the OS sends 64K blocks to 16 x *replication factor*[5] other UnoSys peers in parallel. Depending on the desired safety/liveliness guarantees required for the use of the file, it can be created

[5] Replication factor is the number of times the block is redundantly stored in order to achieve fault tolerance.

for writing with either *strong eventual consistency* semantics for best liveliness at the expense of some safety, or *strong consistency* semantics for guaranteed safety (i.e., single view semantics) at the expense of some liveliness. And because the blocks are written with the replication factor specified when the file was created, they are able to tolerate a well-defined threshold of node failures.

Likewise when an application using the UnoSys API opens a database table and begins reading the records in index order to display to the user, those records are read in parallel from the processor nodes they reside on. Moreover, should a processor node be unavailable, UnoSys will automatically consult one of the other replicated nodes to retrieve the required block, again subject to the consistency model and replication factor specified when the table was created.

Furthermore, the UnoSys node running the application that opens and writes to the above file system and reads the above database need not even have a local persist storage mechanism (e.g., hard drive) itself. There will be many kinds of UnoSys peers, such as *Internet of Things* (IoT) devices, that are capable of running the UnoSys Processor and applications, but will not, themselves, have persistent storage. However, due to the shared view of the same logical single computer abstraction, these nodes will nonetheless be able to access the same shared local file and database systems as any other peer, subject to end-user security permissions. That is, these devices *will* have a persistent local file system and database system as far as the UnoSys applications running on them are concerned, despite not participating in the storage of data blocks themselves.

In short, the UnoSys programming abstraction is significantly simplified as compared to today's dominant client-server paradigm, precisely because the operating system layer (and not the application layer) is handling the complex task of ensuring correctness and availability of the *logically local* file or database table I/O across a massively distributed set of devices all running the UnoSys Processor operating system code.

1.6 Unique Security Challenges of Permissionless P2P Distributed Systems

It is certainly theoretically interesting to entertain the idea of a massive P2P network collectively storing the atomic pieces of every file or table or index that each UnoSys application creates, and to be able to dynamically access them in a massively parallel way such that the whole materializes efficiently and on-demand from the sum of the individual parts. And it is romantic to imagine that this can be done using a single, elegant, logically local computer model with predictable safety and liveliness guarantees, resilient in the face network partitions. However, without addressing the elephant in the room – namely how to secure an open and public distributed operating system – the idea would remain theoretical. The global UnoSys computing vision creates difficult and sometimes unprecedented security challenges that have to be addressed head on in order for it to meet its stated objectives.

For a distributed operating system based on a massively P2P network to become a reality, both the data residing on each peer, and the kernel operating system code running on the peer must be secured with full integrity at all times, even when the peer is under the control of untrusted administrators on

inherently unstable devices. This significantly increases the security challenge over the traditional systems environment where applications "trust" the operating system they run on.

The critical problem to be solved when developing a *permissionless* P2P operating system for the open Internet where any interested person is invited to participate by contributing spare compute, bandwidth and (optionally) storage resources on their computer, is how to ensure that the code running on the node and any end-user data stored on the node, remain both *private* and *tamper proof* at all times. By private, we mean that its contents cannot become known through determined inspection, and by tamper proof, we mean that its contents cannot be altered in any way that would undermine the correctness of the system. This turns out to be a difficult and surprisingly novel problem to solve.

1.6.1 Untrusted System Administrators

The vast majority of unscrupulous behavior that occurs in the context of computing is targeted at *other* people's computers (usually over some network) in order to gain access to some desired resource the hacker wants but does not have access to locally on their own computer. Operating systems are installed by the owners of computer systems for the purpose of running applications locally. Few of those people actually have cause or need to hack the operating system running on *their own* machines. They might do so for the express purpose of better understanding how the OS works so that they may better perpetrate an attack on another computer running the same OS. However, they will rarely hack their own computers for the express purpose of causing harm to themselves because it offers little benefit.

Therefore, interestingly, security is not generally a major concern when *installing* an operating system on a computer. Basically, if you have physical access to a computer, you can install an operating system on it – potentially even overwriting any operating system that is already installed on the computer. Perhaps a bit surprisingly, you are not expected to have *administrator rights* to install an operating system precisely because it is a function of the installation process itself to establish who the administrator is. The critical base premise is that the owner/administrator of a computer is herself fully trusted to act in the best interest of her own local computing environment. Even nefarious owners/administrators need a reliable operating system from which to perpetrate their dishonest activities and so in general do not have need/want to compromise their own computer's operating system.

However, there is a very different security dynamic at play with UnoSys as an Internet-scale open P2P operating system *overlay*. Specifically, *any* interested owner/administrator is invited to install the UnoSys Processor software onto to any of her computers to effectively be a contributing member of a permissionless, massively distributed system. Since the UnoSys Processor code connects to and exchanges data with many other peers in the system, it becomes an interesting if not irresistible attack vector for unscrupulous peer owners to explore for the purposes of exploitation, from the comfort and familiarity of their own hosting computer. Moreover, the data potentially stored on the node belongs to arbitrarily many end-users of the UnoSys system and contains all manner of potentially interesting or valuable information to a node administrator of malicious intent.

This is a unique security problem within the industry since it has been the historical expectation that entire IT departments behind corporate firewalls, or consumers with their PCs behind the firewalls of their home routers, are run by individual owner/administrators trusted to behave in *their* systems' best interest. In contrast, the operational context of the distributed UnoSys operating system must assume that the peers are inherently malicious, exactly because their owners/administrators cannot be trusted.

Solving this difficult problem is critical to the viability of creating a secure reliable OS for the Internet out of a network of potentially hostile nodes controlled by nefarious individuals. It is one thing to secure a system against hackers attempting to compromise remote servers from the "outside". But when you invite these same agents "inside" to join their own devices as nodes to your distributed operating system (i.e., network), you have to address an entirely new vector of security threat. Attacks on the system originating from malicious administrators' very own machines will happen. This reality informs the UnoSys design requiring the system to not only detect this inevitable hostile behavior reliably, but also isolate it and ensure it cannot adversely affect the correctness of the overall system.

1.7 UnoSys – A Fortress of Security

Broadly speaking attacks on the distributed UnoSys operating system are expected on three major axes:

I. Attempts to compromise the end-user data stored on individual peers
II. Attempts to compromise the messages that the operating system communicates between peers
III. Attempts to compromise the UnoSys Processor code running on individual peers

To address each of these attack vectors, UnoSys provides a confluence of proven security technology solutions based on industry best practices. Any single security solution on its own would not be enough to defeat a determined adversary, given that she is the administrator residing over the full resources of her local computing environment in order to mount a focused attack over an extended period of time. However, taken together as a comprehensive set of security measures (i.e., security in layers), UnoSys is able to provide a secure, reliable distributed operating system environment for the open Internet, out of a permissionless collection of inherently unsecured, untrustworthy and potentially malicious nodes.

1.7.1 Securing End-User Data on Peers

As we have already seen, the UnoSys API provides parallel, distributed, stream-based I/O services to realize the high level file or table abstractions within UnoSys applications. The UnoSys Processor, in response to these local requests for data from the application (through the API), efficiently issues requests of its own to the appropriate peers in order to create, read, write and delete the underlying data blocks stored on those peers. In this respect the UnoSys Process proper works at the level of block management.

A *block* – i.e., a piece of a *file*, a record of a *table,* or a page of an *index* – is persistently stored arbitrarily and uniformly on any given peer, itself as a unique otherwise normal file in the host computer's file system. This *block file* consists of its payload and its filename. The block file's filename provides encoded metadata about the block. However, in general there is no way to know which physical block

files are associated with or make up which logical UnoSys end-user files. The challenge then is how to ensure that these block files stored on every node making up the file and database systems are completely secure at all times on the inherently untrusted and unsecured peers on which they are stored.

Since the UnoSys Processor is itself just another piece of software (i.e., application) installed on the hosting peer, it does not and cannot hope to exert any level of reliable control over the operating environment of the computer that it runs on. For example it is fruitless to expect to hide or otherwise keep secret from an administrator of a peer the location where block files are stored. Further, there is no way to prevent a malicious administrator from deleting one or all of the block files. Finally, there is no way to prevent the modifying of the block files by changing arbitrary bits in either its file name and/or its file contents.

Therefore, UnoSys exerts only minimal effort to protect the physical block files stored on any given peer from unintentional harm (e.g., accidental deletion) and otherwise assumes that file blocks can and will be purposely corrupted by determined dishonest agents attempting to undermine the overall system's correctness. UnoSys embraces this eventually and yet is still able to guarantee the integrity of the logical contents of the block files stored on each peer by ensuring that at all times:

1. the logical *contents* of block files can never be learned through low-level inspection
2. the logical *contents* of block files can never be tampered with, without the system knowing
3. the logical *filenames* of block files can never be tampered with, without the system knowing

To address 1 above, UnoSys uses the proven industry standard cryptographic security abstractions of *symmetric encryption* and *hashing* to secure all block files stored on the individual peers. Symmetric encryption randomly transforms a variable-bit string using a symmetric *key* such that it can only be decrypted (untransformed) by agents that also know that same key. It is used to provide privacy of data. Hashing is the process of computing a small, fixed-bit string that probabilistically uniquely represents a very large variable-bit string. Changing as much as a single bit in the very large string will produce a completely different, fixed-bit string. It can be used to verify whether data has been modified.

UnoSys encrypts the content of each block file stored on a peer not once but twice. First, the block file is encrypted using the symmetric key created uniquely for and associated with the logical UnoSys file or database table the block file is a part of. This key is only made available to the UnoSys Processor running the application requesting to create/open an existing logical UnoSys file or database table. This key is part of the file or table *handle* the operating system provides back to the requesting application when the file or table is created or opened. As the application writes data to the file or table stream through this handle, block write requests are encrypted with this key prior to being sent in parallel to the appropriate peers where the respective block files are physically stored. Prior to the receiving peer writing the block file into its file system, the block file is encrypted a second time using the symmetric

key created uniquely for and associated with the peer itself. Only the peer writing the block knows this key.

This approach ensures that, with extremely high probability, no agent can ever learn the contents of an arbitrary block file stored on an arbitrary peer node. First, because there is no way to know through inspection alone which block files belong to which logical UnoSys end-user files, the choice of which block file a dishonest agent will attack is essentially arbitrary. Second, the attacker would have to obtain both symmetric keys where both are not available on the same peer in the general case, and use them in the correct order to decrypt the actual contents of the block file. Third, as part of its key management strategy, UnoSys randomly (yet routinely) independently regenerates both of the above mentioned file and peer node keys and rewrites all affected block files with them. This effectively creates a moving target against long-term brute force attacks to cracking the double encryption scheme.

While the above double encryption process ensures the privacy of the block file, it does not however address issues 2 and 3 from above – protecting the block file from accidental or malicious corruption. A security scheme is required that detects tampering with either the block file's name or its contents. Hashing is used to address this.

UnoSys uses the filename of every block file to store pertinent metadata about the block such as its unique identifier and uses this information to quickly search for the block on disk when serving requests for it. Space is allocated within the filename of each block file to accommodate two separate metadata hashes, one of the block file's contents (i.e., *block content hash)* and the other of the block file's name (i.e., *block name hash)*.

Prior to a peer writing a block file, it first computes the *hash* of its contents and embeds it as the block content hash in the space allocated for it in the block file's name. Then the peer computes the second hash of the block's filename. However before computing this block name hash, a special *placeholder* value for the section of the block filename that holds the computed block name hash is inserted into the block file's name. This placeholder is not a constant[6], but rather carefully computed in dynamic fashion such that it is:

- efficiently computed at the time of writing the block
- efficiently re-computed at the time of reading the block
- pseudo-unique to each block
- not knowable by inspecting any aspect of the block (i.e., the block file name or its contents)
- not knowable by inspecting the UnoSys operating system source code
- computes the same on all replicas

[6] If a known constant *placeholder* was used it would be possible for a nefarious agent to change some aspect of the block file name and then re-compute the hash of the block filename using the constant placeholder and re-embed it into the block filename, effectively succeeding in tampering with the block filename without UnoSys knowing.

The block filename containing this specially computed placeholder is then hashed to compute the block name hash and the result is used to replace the placeholder value in the filename to produce the completed filename of the block file. Finally the peer writes the block file with this completed filename.

Upon a subsequent access request for a block, the UnoSys Processor looks up the block file by unique identifier and removes the block name hash from its filename and sets it aside, replacing it with the original placeholder used above. Then the hash of this augmented filename is computed and compared to the set aside block name hash to ensure they match. These will not match if so much as a single bit of the block file's name has been changed since the block filename was computed using the above process just prior to the time of last write. Assuming the file name has not been tampered with the peer then computes the hash of the contents of the block file and compares that to the block content hash embedded in the block file's name. These will not match if so much as a single bit of the block file's contents has been changed.

If either of these checks fail, or if the block file cannot be found at a peer it is expected to reside on (i.e., the block file has been deleted), then the system knows that the peer has been compromised and returns an appropriate error indicator to the peer making the original block access request. Should a requesting peer receive such an error indicator in response to a block access request, it reports the faulting peer to UnoSys as compromised (more on this later), but otherwise tolerates the failure by sending the same block access request to a different peer containing a replica of the block. Therefore, while it might be entirely possible for a dishonest agent to alter (or even delete) the contents of end-user block files stored on an arbitrary peer, it would be impossible to do so without the system knowing about it when attempting subsequent access to the corrupted block files.

In this way the UnoSys Processor ensures that every block file persisted on every peer within the distributed system is at all times fully secured in that its contents can be neither discovered, nor tampered with without the system knowing. Hence, UnoSys maintains its safety properties with respect to end-user data it stores by guaranteeing the complete privacy and complete correctness of the data stored in plain sight across all of its potentially hostile peers.

1.7.2 Securing Data Passed Between Peers

Securing persistently stored data at *rest* on a UnoSys peer is only one aspect of data security. Data must also be secured while in *motion* as it is passed between peers. In order to secure all data messages exchanged between peers, UnoSys uses the same proven symmetric encryption and hashing cryptographic security abstractions it uses to secure end-user data at rest, along with industry standard *digital signatures* based on *public/private asymmetric encryption*. Digitally signing a message allows the receiving peer to verify the message was in fact sent by the peer that claims to have sent it. This requires the sending peer to know its own private key and the receiving peer to know the sender's public key.

UnoSys secures a message in motion between peers by first digitally signing it using the private key of the sending peer. The peer uses its private key to sign a hash of the message payload to create the

message's digital signature. It then concatenates this digital signature with the payload of the message and the result is then *encrypted* using a *session key* – a symmetric key created uniquely for and associated with the communication channel opened between the two peers. The symmetric key is generated on the peer initiating the creation of the communication channel, and is shared with the other peer as part of the initial handshake protocol[7]. In this manner, both peers of the channel pair (and only those peers) share the session key. The resulting signed and encrypted message is then sent to the other peer through the established channel.

Upon receiving the message, the receiving peer decrypts the message using the shared symmetric session key. It then attempts to verify the enclosed digital signature of the message to ensure it was in fact sent by the peer at the initiating end of the channel[8]. The result of the digital verification process is compared to the recomputed hash of the message payload and, if they match, the message is known to have originated from the peer that sent it and not tampered with. In this manner UnoSys guarantees the safety of messages pair-wised exchanged between any two peers, protecting against so-called man-in-the-middle and/or arbitrary message forgery attacks.

UnoSys employs this secure messaging technique for both message requests and their responses. It uses well documented and fully vetted cryptographic libraries to ensure that each data exchange between nodes is verifiably safe. However, no mention has yet been made of how the shared secret symmetric keys and public/private asymmetric keys used extensively are themselves secured. This then shifts the scrutiny in implementing secure messaging between UnoSys peers to the key management issue, since the overall cryptographic security of the system will only be as sound as the management practices employed to protect the keys. But before key management security can be explored below, we first have to introduce the extraordinary measures UnoSys undertakes to protect the UnoSys Processor code that runs on each node.

1.7.3 Securing UnoSys Processor Code

As has already been discussed, a reality of today's Internet is that computers cannot be relied upon to behave honestly. Software running on these computers can be compromised in Byzantine ways – the most egregious being when it comes under the control of a nefarious agent who intentionally changes its behavior to act in arbitrary ways. Dealing with this issue is especially problematic for open-source projects where anybody can download the source code, study it, alter it in any imaginable way, and then rebuild it to create a new unofficial version that can be run with new and arbitrary behavior for any intended purpose.

[7] UnoSys protects the *initiating* handshake protocol messages used in establishing a connection between peers by encrypting the messages with a separate so-called *computer symmetric key* that every peer implicitly knows. This key is periodically regenerated and redistributed to every peer in order defend against brute force attacks designed to guess it.

[8] During the creation of the communication channel between two peers, the public keys of each peer are freely exchanged as part of the initiating handshake protocol, ensuring each peer knows the public key of the other.

One major category of open source software projects is *libraries* of functionality intended to be incorporated into larger applications. Another major category is entire *platforms* (e.g., Linux operating system, MySQL database, etc.,) that are intended to be used as is or in rare instances customized for specific scenarios. In both cases worry over unscrupulous agents modifying the code is tempered with the knowledge that for any advantage to be had by the hacker, he would still have to find a way to distribute their poisoned versions to people in order that they run them within their environment. Every day, phishing exploits cause unsuspecting *end-users* to inadvertently install infected *binary* applications containing viruses, which do immeasurable harm. However, it is a lot harder for malicious agents to distribute poisoned versions of open source projects because the *users* of these projects are generally experience IT professionals and typically either use official/authorized open source project binaries, and/or compile the binaries from official open source project distribution sources themselves.

However, once again a permissionless P2P operating system such as UnoSys represents a unique security context challenge. Since the UnoSys processor software is installed *locally* and a nefarious agent has complete administrative control over his personal computer, the distribution obstacles from above are no longer an issue. Hence, if provided with source code, a nefarious agent could produce a custom version of the UnoSys Processor code, and install it on his own computer. Such a version of the software would be under his complete control and could easily be made to behave in any possible Byzantine manner. In this way the agent could inject a poisoned node into the UnoSys P2P network and have it undermine the correctness of the overall system as a whole in any imaginable way.

Even more worrisome, however, is that source code to the UnoSys software would not even be required for an experienced hacker with ample time on his hands to compromise the correctness of the system. If the UnoSys project did not provide source code, but instead provided only an official binary distribution that users installed in the form of classic .EXEs *executables* and .DLL *libraries*, it would still be possible for malicious users to compromise the integrity of the overall system, precisely because they completely control the environment those binaries execute in. This is a critical subtlety that virtually all existing P2P software distributions overlook, and is a gapping whole in their security model.

.EXE and .DLL binaries, especially those that run on the popular Java or .Net *runtimes* (i.e., the JVM – Java Virtual Machine, or .Net CLR – Common Language Runtime), are vulnerable to manipulations. Java and .Net are increasingly more popular for large project development for all of the productivity benefits they bring to the developer – type safety, automatic garbage collection, extensive libraries, cross platform support, and a developer talent pool in the tens of millions, to name just a few. Furthermore, the JVM and CLR runtimes are extensively configurable, designed to be customized for many *hosting* scenarios.

However, once again we see that the base assumption for all of these configurations and all of these scenarios is that the administrator is trusted to act in the best interest of the hosting computing environment they manage. It is an historic given that he will act in a manner that keeps the hosting environment safe. This, as a general rule, is true for the *Enterprise* (i.e., private networks) as the administrators are paid to be this very trusted agent, working within a presumably secure private

network designed to keep the good guys in and the bad guys out. The code that runs within a private network is therefore itself "trusted" by the very fact (and only by the fact) that a trusted administrator installed it within a trusted runtime environment (i.e., JVM or CLR).

However, as we have already seen, this base premise of trust is nonexistent in the open, permissionless security context of an Internet scale P2P application such as UnoSys because the administrator of the machine contributing a node to the P2P network cannot be trusted to host the Unosys Processor code in a manner that is in the best interest of the overall system. The reality is that a motivated hacker with time on their hands and complete control over the machines that they administrator would be able to relatively easily compromise a Java or .NET binary only distribution of the UnoSys Processor code in Byzantine ways.

Stated simply, a binary file distribution of *any* software project is fundamentally insecure because there is always a way to reverse engineer it on the machine it executes on. This is easy if not trivial for Java .Jar files or .Net .Exe/.Dll files with decompiler tools that are readily available. However, perhaps more disturbing, is that for many hacks, reverse engineering the code is not even necessary to gain control over it.

Take for example a JVM installation. As mentioned, the runtime environment of the JVM is extensively configurable by an administrator of the machine it runs on. The JMV can be configured to load a custom **SecurityManager** class to change global security policy for the entire JVM runtime environment. In short, the administrator can change fine grained security settings of the environment in any way she likes by simply authoring her own **SecurityManager** class and configuring the JVM to load it at start up. For example, the policy to change the visibility of class fields can be controlled to allow access to the private member fields of classes. This simple configuration means any class that might have an interesting private member field named something like **Key** will be available outside the class to any caller, when it should not be. This would completely undermine the visibility scoping of class member fields that developers understand and rely on to encapsulate their data abstractions. Another more global security policy change could be configured to ignore default security checks altogether, checks that Java developers rely on to authorize and authenticate the code execution context[9].

An even more powerful tamper vector would be for an administrator to supply her own **ClassLoader** class to the JVM. This is a well understood configuration extension that allows JVM implementations to control where classes are loaded from (for example from a database or over the network rather than from .Jar files off disk). Essentially, a custom supplied **ClassLoader** allows an administrator to control where code for any Java application comes from. This empowers any JVM administrator with the ability to specify their own **ClassLoader** to provide a convenient way to effectively substitute any desired class

[9] NOTE: The .Net CLR provides similar global environment controls providing equivalent .Net hosting configuration capabilities.

in the software distribution with an interface equivalent user-supplied class[10]. This mechanism, in combination with a relaxed security policy as described above, would allow entire class implementations within the original software distribution to be completely replaced with custom dynamically loaded versions authored by the administrator, allowing the behavior of the original class to be altered in any imaginable way.

Hence, the reality of the unique security context of a P2P UnoSys operating system is that providing a Java or .Net binary distribution of the software to a malicious agent to install on their own computers as administrators is simply not viable from a security perspective as it would inevitably lead to compromised (i.e., unsafe) nodes running in the network in Byzantine ways, undermining the stability of the overall system. UnoSys uses a novel, sophisticated approach to its software distribution to address this seemingly dead-end security result.

1.7.3.1 UnoSys Dynamic Code Delivery

The major technological objective of UnoSys is to design a reliable, distributed P2P operating system at Internet scale. This necessarily requires running trusted, secured (i.e., tamper proof) UnoSys code on a network of potentially untrusted, unsecure and potentially malicious computers. In order to meet this objective UnoSys must be able to ensure with very high probability that the code running in any of its nodes is "correct". That is, it is running the official, fully vetted code that is part of its distribution and that it has not been tampered with in any way. This is not possible using a traditional "downloadable" software distribution approach, since any such code provided in file format (i.e., source code or binaries) is subject to security compromises in the ways previously discussed. UnoSys addresses this issue by eliminating the traditional "file" distribution of its software altogether by creating a *fortress* for its code to be dynamically downloaded directly into, in order to run on every node.

Sandbox is a term used to describe a process that hosts untrusted code, but limits its access to the computer that it is running on. For example a sandbox might dynamically load a foreign .DLL and execute it, but not before ensuring it has limited-to-no access to local computer resources such as the hard drive and the network. Sandboxes are designed to protect the *computer* on which they run from the *code* running within it.

In contrast, UnoSys needs to create a *fortress* on a computer which does everything a sandbox does to protect the computer from the code running within it, but also protects the code running within it from the computer environment itself (i.e., from nefarious administrators). A custom built, tamper-proof *virtual machine* environment is required to create a fortress and this is exactly what UnoSys provides for each of its nodes.

The UnoSys operating system proper is written in C# and runs in the .Net platform. .Net was chosen over the very capable Java platform because it was found to have a better *hosting* model for the unique

[10] NOTE: The CLR does not provide a configurable **ClassLoader** facility in the way the JVM does. However, it does allow for the configuration of where entire .DLLs are loaded from, effective making the same kinds of dynamic code substitution hacks possible.

security context that is required for a permissionless distributed P2P operating system project. Whereas the default JVM is extensively configurable, .Net goes one step further and allows you to by-pass the default (yet configurable) CLR runtime, and create a completely custom host for the CLR from within your own software process and lock it down in a manner that no one, including the administrator of the machine, can change.

The **UnoSys.Processor.exe** that runs on every UnoSys node is itself a small C++ application that does nothing but host a custom .Net CLR runtime implementation. Upon startup, the UnoSys.Processor.exe loads a copy of the standard .Net CLR and then sets about configuring it by locking down all of its settings and disabling the ability for them to be changed. It then connects to the Internet and securely downloads the .Net **UnoSys.Kernel.dll** C# assembly (among others) directly into the address space of the newly configured CLR host and starts it running. This .Net assembly represents the kernel UnoSys operating system code proper and the act of downloading it and running it amounts to bootstrapping the node execution in a custom .Net virtual machine runtime context.

In this manner no actual UnoSys C# *operating system code* ever exists on the disk of the computer that is hosting it. In fact the entire operating system proper[11] is dynamically loaded across the Internet every time the UnoSys.Processor.exe is started. With this general approach, the two glaring security problems described above are addressed:

1. The CLR runtime *host* process (UnoSys.Processor.exe) is not configurable by the administrator and so cannot have its functionality (i.e., its security policy or where code is loaded from) altered in the standard ways provided by default .Net runtime
2. The actual running code of the UnoSys operating system is never available to the administrator in the form of a file on disk.

However, securely loading code dynamically over the Internet is not new. Web browsers have been doing it for decades. And while it is more secure than loading the code off the local disk for the foregoing reasons, it is still susceptible to security attacks of a different nature – those that focus on accessing the code where it resides at runtime – inside the memory of the running process that hosts it. For example, a malicious user with full administrator rights to their own machine could easily attach a debugger to the running UnoSys.Processor.exe and gain the ability to inspect the low-level assembly code as it runs. Alternatively, it is trivially easy to "dump" a running program's memory layout to a disk file and then study it at leisure offline. The UnoSys system addresses this type of attack through a strong notion of node identity.

1.7.3.2 Processor Identity

Rather than the traditional software approach of creating an application and giving users a bit-wise identical *copy* of it to run, UnoSys takes the approach that the software running on each node of the P2P network is unique to that node only. Many software vendors embed unique serial numbers, etc., in

[11] The entire C# operating system code weighs in at approximately 0.5 megabytes in size at the time of this writing.

their binary distributions to make them unique. But UnoSys takes the notion of uniqueness among distributions to a significantly more sophisticated level. As a result the security it achieves is unparalleled.

To this end, UnoSys defines a node *manufacturing* process that generates unique, cryptographically signed **UnoSys.Processor Installation** packages containing a uniquely generated, obfuscated, compiled, encrypted and digitally signed UnoSys.Processor.exe executable. No two UnoSys.Processor installation package files are alike and these installation files are designed to self-destruct after they are installed, ensuring they can never be installed on more than one machine, or reinstalled on the current machine. The result is that each and every UnoSys peer installs a unique piece of software that acts as a host for the UnoSys distributed operating system. Moreover, and significantly, the UnoSys system as a whole can uniquely identify each and every node making up the system. This forms the basis of a reputation system whereby every node is monitored for its behavior and can be disciplined by the system as a whole (e.g., quarantined) if it is found to be misbehaving.

Nodes are manufactured by a trusted authority (more on this later) in advance and stored until they are requested by a user. Anyone wanting to install UnoSys on their machine simply downloads a unique UnoSys.Processor installation package instance from a website and installs it[12]. This installs the small C++ UnoSys.Processor.exe host that boots itself by authenticating against the UnoSys *Code Delivery Service*[13] (using its unique node ID given to it by the manufacturing process) to request the download of UnoSys.Kernel.dll assembly code instance paired specifically for it and then hosts it in CLR Fortress environment. The UnoSys.Kernel.dll code itself is also uniquely generated, obfuscated, encrypted, compiled and digitally signed by the same manufacturing process that generates the UnoSys.Processor installation package file and the UnoSys.Processor.exe that it contains, in such a way that the UnoSys.Kernel.dll will only load in the UnoSys.Processor.exe instance that it is cryptographically paired with.

Moreover, the UnoSys system ensures that each and every time the UnoSys.Processor.exe is *restarted*, it dynamically loads a completely newly compiled, obfuscated, encrypted and digitally signed version of the UnoSys.Kernel.dll created specifically for that running host *instance*. While functionally the same as all other UnoSys.Kernel.dll instances, the UnoSys.Kernel.dll that a particular UnoSys.Processor.exe host instance loads is encrypted such that it can only be loaded by that host execution *instance*.

This strategy ensures that:

1. The Code Deliver service verifies that the request to boot is coming from a valid/verified node (i.e., one known to be manufactured, and uniquely installed)

[12] Initially, UnoSys.Processor installation package files will be available from a centralized website for bootstrapping purposes. However, after the UnoSys network reaches critical mass, the operating system itself will manufacture the unique installation packages and store them within the UnoSys Computer directly.

[13] Initially, the UnoSys Code Deliver service is centralized for bootstrapping purposes. However, after the UnoSys network reaches critical mass, the service will be subsumed by the operating system proper and the service will be provided in a decentralized manner by the OS itself.

2. The kernel software loaded into the node is verifiable UnoSys code that can only be loaded and run in that specific node instance
3. The in-memory layout of the kernel code dynamically loaded in a Unosys host instance is entirely different on each and every execution
4. All communications between node and UnoSys Code Delivery service are fully encrypted and signed by both parties
5. Kernel code sourced from the UnoSys Code Delivery service is uniquely generated, obfuscated, compiled, encrypted and digitally signed (during manufacturing) and installed directly into running memory of the node for which it is paired, and never touches a disk.

UnoSys goes further still by building in runtime checks within its UnoSys.Processor installation package, its UnoSys.Processor.exe host and its UnoSys.Kernel.dll operating system software to make it even more difficult to learn about the software running within the fortress environment created on every UnoSys node:

- The software continually checks if it is running under a debugger and if so found, will automatically report to the greater system that the node is running in a potentially hostile environment (as identified by its IP Address, peer node ID, and computer serial number) before shutting down. Repeated infractions of this type will result in the computer being blacklisted as a suspicious UnoSys node host, and the system refusing to install future UnoSys.Processor installation packages.
- The software will periodically (i.e., every few hours) automatically restart itself in order to guarantee a fresh newly keyed UnoSys.Kernel.dll is loaded – ensuring its software is always up to date including the latest updates, as well as that the layout of that software in memory is continuously unique.

While these security measures do not prevent an administrator from dumping the memory contents of a running Unosys.Processor.exe process to disk for examination, it does ensure that they are seeing a completely different layout of low-level assembly code each time, bearing no relation to the layout they would observe on any other node they may be analyzing on another machine, let alone the very next run of UnoSys.Processor.exe on his current machine. This means that any information gleaned from efforts to reverse engineer the software via off-line crash dumps cannot be leveraged on the next running of the software.

Further still, the extensive UnoSys P2P node software security discussed above had to also be extended to cover the installation process itself. As a result, the installer was designed to use the same technology as described in 1)-5) above in that its core C# installation code is also uniquely created for each installation and dynamically loaded at runtime from the UnoSys Code Deliver service into a custom C++ CLR runtime virtual machine for execution, designed specifically for installing the UnoSys software. Moreover, in addition to the tamper resistant precautions outlined above, the installer has further security enhancements to address expected attack vectors during the installation process itself:

- The actual UnoSys.Processor installation process is itself monitored (checking if it is running under a debugger, or checking if it takes more than a few seconds to run) and will report suspicious activity before self-destructing the UnoSys.Processor installation package
- The installation process must be run within a set amount of time after download or it will expire and self-destruct
- The installation process must be completed in under 4 minutes
- The installation process can only be run on a single computer
- If the software is subsequently uninstalled after installation, the original installation package cannot be used to re-install the software (i.e., the user must download a fresh install package).

All of the foregoing security measures ensure that the installer is used only as intended and that the UnoSys Code Delivery service maintains detailed knowledge/crosschecks between what nodes have been downloaded and what nodes have actually been installed.

Taken collectively, these significant security measures combine to ensure that a UnoSys.Processor.exe *fortress* process not only protects the computer from the UnoSys.Kernel.dll operating system code that runs within it, but also protects the UnoSys.Kernel.dll operating system code from being tampered with from the hosting computer environment. This provides the critical "safety" guarantees for the overall UnoSys distributed operating system because it guarantees that code always runs "correctly" because it is at all times tamper proof – a result that cannot be achieved via traditional open source and/or binary software distribution models.

In summary, this unique security context requires:

1. A custom built, tamper-proof, virtual machine runtime environment
2. That all operating system code running in the virtual machine be dynamically loaded at runtime (i.e., never resides anywhere on the computer's disk at any time)
3. That all operating system code be generated, obfuscated, compiled, encrypted and digitally signed uniquely for each peer
4. That all operating system code be re-generated, re-obfuscated, re-compiled, re-encrypted and digitally re-signed each time the UnoSys.Processor.exe is restarted on a peer

In this way, UnoSys has created a sophisticated, state-of-the-art solution to the issue of installing and running tamper-proof code on a distributed P2P network of inherently untrusted and unsecured computers. Importantly, this renders an entire class of Byzantine faults (i.e., those achieved by modifying code behavior) as a practical impossibility in the UnoSys system.

1.7.4 Byzantine Fault Tolerance Revisited

Previously, the notion of asynchronous Byzantine Fault Tolerance (aBFT) was introduced as the strongest fault tolerance guarantee a distributed system can achieve. It describes a systems model guaranteed to be tolerant to the widest possible range of reasons for node failure including:

1. Nodes crash

2. Nodes are unreachable (i.e., stop responding)
3. Nodes are reachable but Messages are delayed (possibly indefinitely)
4. Messages arrive out of order
5. Messages are dropped (i.e., never sent)
6. Messages are sent multiple times
7. Messages are invented out of thin air
8. Messages are eavesdropped on but otherwise not altered
9. Corrupt Messages are sent
10. Legal but wrong Messages are sent

The three (3) main extensive Unosys security measures described above directly address items 7 to 10 of the foregoing list:

I. The security measures to protect end-user data stored on individual peers addresses item 8 on the above list
II. The security measures to protect messages that the operating system communicates between peers addresses item 9 on the above list
III. The security measures to protect UnoSys Processor code running on individual peers addresses items 7 and 10 on the above list.

In order to address items 1 to 6, UnoSys is designed for and assumes a *synchronous systems model*.

1.7.4.1 Synchronous Systems Model through FIFO Links

UnoSys simultaneously addresses the issue of *asynchronous*-BFT and *crash detection* by requiring all node-to-node communications to support message order delivery semantics and bounded timeouts. This is achieved by basing all internode communications on the relatively new WebSocket protocol[14], an efficient, firewall and proxy friendly communication protocol standard that runs over HTTP on all targeted UnoSys node environments.

UnoSys runs the WebSocket protocol over port 80. Since HTTP itself runs on top of TCP over IP, WebSockets inherit the error correcting, order delivery semantics of TCP over IP to achieve efficient two-way duplex FIFO (i.e.; messages are sent in order *first-in*, *first out*) communications between nodes.

Each UnoSys node is both a WebSocket client and a server, capable of sending exactly one message at a time, in either direction, simultaneously. Modern servers can handle hundreds of thousands or even millions of unique simultaneous WebSocket connections, which will support a P2P distributed systems architecture at Internet-scale. Finally, because the WebSocket protocol inherently supports *TimeToLive,* configurable timeouts can be established on each connection, which will reliably detect when a node is

[14] Support for the stronger *asynchronous* Byzantine Fault Tolerance (aBFT) model becomes moot and is reduced to mere (synchronous) BFT given a synchronous systems model that assumes strong persistent connections using WebSockets.

no longer responding. This enables UnoSys to have reliable node failure detection within bounded time, allowing it to compensate accordingly by resending the failed node request to other node replicas.

With FIFO links and configurable timeouts, WebSockets provide reliable time-bounded node failure guarantees reducing the main *asynchronous* BFT issue at hand to the more tractable *synchronous* BFT problem. This synchronous systems model assumption and implementation ensures that UnoSys can address the remaining Byzantine fault categories from the list above (i.e., items 1 to 6), since the system is able to detect and tolerate them all. Simultaneously, assuming a *synchronous systems model* with FIFO links addresses many objections to asynchronous distributed system design that can't assume bounded time node crash detection, thereby allowing for the use of fully vetted yet significantly less complex *synchronous* distributed algorithms to be used in the implementation of UnoSys.

The synchronous systems model guaranteed by the WebSockets implementation of FIFO links fully addresses items 1 to 6 on the list above of ways a node can be Byzantine faulty. Hence, it has been shown how from a single peer node perspective UnoSys will provide a BFT systems model on which to build a reliable, permissionless, P2P distributed operating system for the purposes of supporting an Internet-scale single Computer abstraction. In the ways described, UnoSys makes it theoretically and practically impossible for a single node or a collection of colluding nodes to compromise the safety and liveliness of the UnoSys network within a configurable number of tolerated faulty nodes. However, for completeness, a discussion is warranted on how UnoSys will defend against inevitable distributed denial of service (DDoS) attacks originating outside the network.

1.7.4.2 Dealing with DDoS

All networks are targets for *Distributed Denial of Service* (DDoS) attacks. DDoS attacks are particularly sinister because, in contrast to *hacking* attacks which are often motivated by an opportunity to profit from the network (e.g., gain control over a system to steal credit card information, etc.), DDoS attacks are largely designed to simply cause damage to the network – ultimately, as its name implies, to cause the system to be unable to provide its intended service. This is somewhat analogous to the difference between a thief and a vandal, with the former typically motivated by self-interest profit and the latter typically motivated by other reasons such as revenge or wishing to make a political statement[15]. Irrespective of motivation, DDoS attacks focus on affecting the *liveliness* of the system it is perpetrated against, for the ultimate purpose of preventing it from making progress. The technical goal of the attack is to cause the system to stop functioning within its designed tolerance levels.

Application Layer attacks are a common class of DDoS attack which seek to overwhelm specific features or services of a system. UnoSys is inherently well suited to defend against these attacks, because, unlike traditional HTTP web services, UnoSys does not expose a centralized endpoint that clients call – the applications that run on the UnoSys operating system have no notion of a network, making application network attacks impossible.

[15] NOTE: It is possible that profit may be the ultimate motivate of the DDoS attack as well (i.e., a perpetrator could plan to extort a ransom payment from the proprietor of a website that it is attacking).

Instead, UnoSys only supports pair-wise communication between its nodes from code deep within the operating system itself. Peer nodes must already internally know the endpoints of the nodes they wish to communicate with *and* receive communications from – i.e., the set of endpoints is a closed and relatively small membership. Hence, it is relatively easy for UnoSys nodes to filter requests from unknown nodes.

Of course, more advanced attacks can spoof sender addresses to make requests seem like they are coming from known endpoints. However, the attacker messages would then assuredly fail digital verification if they were not legitimately sent from the *correct* node in the network that claims to have sent them, and therefore would be rejected on that level long before they were able to "enter" the system via the target node. Therefore, the previously presented strong message exchange security, in combination with the considerable *fortress* security infrastructure UnoSys uses to ensure tamperproof code, makes it practically impossible for a malicious agent to counterfeit legitimate messages either from inside or outside the network.

Another popular DDoS attack known as Amplification is achieved through DNS exploits that most traditional websites are susceptible to because virtually all of them use DNS to translate well-known domain names (e.g.; Google.com) to their public endpoint network addresses. This kind of attack is not a concern to UnoSys because all pair-wise peer communications within the network are done directly with IP Address and no DNS lookups are used or required.

In general, defensive responses to DDoS attacks typically involve the use of a combination of attack detection, traffic classification and response tools, aiming to block traffic that they identify as illegitimate and allow traffic that they identify as legitimate. The design of UnoSys makes it well suited to implement these kinds of defenses, both locally on each node, as well as collectively across all nodes, because UnoSys nodes are collectively self-organized into a single global autonomous system[16].

Thus, as individual network traffic patterns suggest a peer is under attack, it can quickly share that information with its neighboring UnoSys peers. As such, it is possible for one peer to learn from another peer about mounting sophisticated DDoS attacks elsewhere in the network. The natural spreading of this information throughout the UnoSys network allows the operating system to autonomously mount a strategy to defend itself against the attack and dynamically implement it in near real-time. Moreover, such responses would not in theory be limited to defense only. Over time, the unique, fully distributed, yet logically autonomously centralized design of the UnoSys operating system would allow it to theoretically go on the offensive and implement aggressive counter measures of its own, such as honeypots designed to lure attackers into harmlessly attacking decoy target resources in order to discover their identity.

[16] Further discussion on the governance model for the World Computer can be found in the companion document "***A Vision for a New Dignified World Computer Economy***" available on the World Computer Organization website (www.WorldComputer.org).

Even though UnoSys makes DDoS more difficult, the nature of the way the Internet works makes it impossible to prevent a determined malicious agent from obtaining the IP Addresses of a sizeable group of UnoSys peer nodes (through prolonged and patient but otherwise standard network packet sniffing tools and techniques) and flooding them with garbage message traffic from multiple locations outside the network (e.g., from a botnet). UnoSys would be able to guarantee the *safety* of the system at all times under such an attack. However, the *liveliness* of the system could be compromised depending on the sheer size of such a coordinated attack.

Fortunately, the intended scope of the UnoSys operating system in implementing a planetary scale single Computer abstraction works to its favor and suggests that once the operating system achieves critical mass, it would be theoretically impractical to mount a DDoS attack big enough to bring it down in its entirety, in the same way the Internet itself has never been taken down in its entirety. UnoSys is a state-of-the-art distributed system implemented with proven distributed algorithms design to tolerate DDoS attacks causing network partitions and yet continue to make progress. Therefore, it will theoretically be able to locally self-detect and then globally self-adjust to the attacks in order to mitigate them and therefore protect the system's liveliness properties.

Another argument for why UnoSys will be less susceptible to large scale DDoS attacks than typical government or corporately controlled Internet properties is that the digital anarchists who tend to perpetrate such attacks for political reasons will hopefully be supportive of the ideals that the UnoSys project stands for. These people are arguably more likely to understand the motivation behind the project and to recognize the benefit a massive Internet-scale world computer open to all and controlled by the commons will bring to the world. The expectation is that many of these often passionate people will chose to support, promote and directly contribute to the initiative in order to foster a digitally freer future, controlled by a community through a democratic oversight process[17], rather than by an oligopoly of few big tech companies or government agencies.

Finally, perhaps the most powerful way in which UnoSys can defend against large scale DDoS attacks is to provide an economic incentive for people to take the considerable computing resources and energy it would otherwise require to mount such attacks against UnoSys, and redirect them to be used legitimately on the network in order to generate real monetary profit[18].

1.7.5 Cryptographic Key Management

As mentioned previously, the overall cryptographic security of the system is only as robust as the management practices employed to protect the cryptographic keys themselves. Here again, UnoSys has been carefully designed to ensure secure key management in the face of untrusted nodes.

[17] More information regarding the World Computer governance model can be found in the companion document "***A Vision for a New Dignified World Computer Economy***" available on the World Computer Organization website (www.WorldComputer.org).

[18] A detailed discussion of the economic incentives for using the World Computer can be found in the companion document "***A Vision for a New Dignified World Computer Economy***" available on the World Computer Organization website (www.WorldComputer.org).

As previously mentioned, UnoSys is written in .Net. Today the modern .Net Core runtime is cross platform, supporting the Windows, Linux, MacOS and Android environments, among others. Each of these hosting environments natively implements cryptographic key management entirely differently. By design, UnoSys does not store cryptographic material in the traditional key stores of the nodes' hosting computer (e.g., the Windows Certificate store), because these stores are under full control of potentially malicious administrators and cannot therefore be trusted. In order to address this particular aspect of the unique security challenges introduced by a permissionless p2p operating system security context, UnoSys must provide its own custom key management services built from scratch.

To address this, UnoSys leverages its secure, tamperproof code delivery and fortress architecture to protect the cryptographic material used to secure the system. Recall that the UnoSys code is (re)generated, (re)obfuscated, (re)compiled, (re)encrypted and digitally (re)signed for each unique UnoSys.Processor.exe execution. In addition, the system takes the opportunity to (re)generate fresh keys for all symmetric and asymmetric cryptographic purposes wherever it makes sense, ensuring they are continuously being refreshed as needed. These keys are (re)generated by the system-trusted UnoSys *manufacturing* process, and are embedded directly into the UnoSys.Processor.MSI, UnoSys.Processor.exe and UnoSys.Kernel.dll code binaries as appropriate. These generated binaries and the keys they contain are then dynamically loaded into UnoSys.Processor.exe fortress upon its next startup and are therefore never stored on disk.

This approach ensures that the core security layer of UnoSys is fundamentally governed by the safe generation and storage of essential cryptographic keys, along with the automatic expiration and revocation of, and subsequent replacement, with new, temporal keys, all performed in a self-managing manner on a continuous basis.

1.7.6 Self-Organization

To this point, little has been said about how the UnoSys P2P operating system will organize itself. The goal is for UnoSys to be permissionless – open to anyone interested in contributing their spare computer resources. However, from an organizational point of view, such a P2P system could be implemented in many ways ranging from a centralized architecture such as that taken by Napster to a fully decentralized architecture taken by Gnutella.

P2P systems have been studied extensively in the computer science literature over the last three (3) decades. As such, the technology has matured into different sub-fields depending on the desired design requirements for the P2P system. As is usual in the realm of software and algorithms, tradeoffs are made with respect to design and implementation depending on the functional requirements of the services that will ultimately run on top of the P2P system.

For example, applications ranging from file-sharing, reliable multicast, redundant storage, or group membership, are all "services" that can be built on a P2P system. Furthermore, the type of P2P system you would use to implement any of these services might be different depending on the characteristics of

the environment; i.e., whether anonymity between peers is required, whether nodes have global awareness of each other, and/or whether the operational environment itself can be trusted.

UnoSys implements its P2P solution based on two separate organizational architectures:

1. A *structured overlay*
2. A cluster

A *structured overlay* connects a very large set of nodes in a self-organizing way into a network. Each node of the network is connected only to a small subset of neighboring nodes that it knows about. Scale is attained in such an overlay because millions of nodes can be loosely connected in a global way, without each node being aware of all other nodes in the network. Instead, each node is aware of a very small number of neighbor nodes and *routing* a message from an arbitrary source node to an arbitrary target node is achieved by communicating between pair-wise neighboring nodes until a path can be found between the source and destination nodes. Such a path can require many logical "hops" from one node to another to materialize, but eventually the message will be delivered in the absence of network partitions.

In contrast, a *cluster* is a network of computers that know about each other. That is, a cluster has global awareness because it knows about all other nodes in the network. In a clustered network, each node is connected to every other node which means any two nodes can communicate directly with a single logical "hop" assuming full network connectivity.

Each of these networking architectures has relative strengths and weaknesses. Structured overlays scale to very large networks, whereas cluster networks do not. However, clustered networks provide efficient single logical hop communications between nodes, whereas structured overlays generally do not. And structured overlays provide more flexible support for the ad hoc joining/leaving of nodes to/from the network than clustered networks, which generally assume a fixed node membership.

In order to gain the best of both worlds, UnoSys utilizes both of these structures in its implementation. Initially, UnoSys nodes first join an open structured overlay network. Nodes joining the UnoSys P2P network can immediately begin to be utilized by the applications that run on top of the operating system. That is, the nodes can immediately begin to service *client* requests from the applications.

However, these nodes are not yet allowed to participate as *servers* on the network in that they are forbidden to store and subsequently serve up end-user data in the form of I/O file blocks previously discussed[19]. Later, after a node has established a well behaving *reputation* over time, it will become a

[19] A discussion of how UnoSys avoids the well documented social phenomenon within P2P systems known as the "tragedy of the commons" where end-users free-ride by using the client-side aspects of the system without themselves contributing any server-side resources required for the system to work, is given in can be found in the companion document "***A Vision for a New Dignified World Computer Economy***" available on the World Computer Organization website (www.WorldComputer.org).

candidate to be included in one or more of the more exclusive replicated clusters used for specific UnoSys services such as storage or consensus, which requires elevated server capabilities.

In this way, a deserving node's participation within the UnoSys operating system evolves over time from a limited client-only role to a more capable client-and-server role. The decision as to when a UnoSys node has established enough reputation that it can be promoted to a full client-and-server role, and reap the benefits and rewards of that status, is made globally by the UnoSys system as a whole, without human intervention.

1.8 UnoSys Technology Summary

UnoSys is a modern, reliable, permissionless, distributed peer-to-peer operating system designed to implement a World Computer. UnoSys abstracts away all notion of a network, exposing only a significantly simplified *single computer* abstraction to developers writing applications for it. These applications are simpler to write and reason about than their client-server architecture equivalents, and can be authored by programmers of any skill level in any modern programming language with no loss of expressive power. Furthermore, simply by utilizing the UnoSys operating system's modern stream-based file and database I/O services, any application written for it automatically gains full distributivity and decentralization at Internet scale.

The UnoSys *systems* and *security* model is designed to achieve the highest standard in distributed systems fault tolerance – Byzantine Fault Tolerance. Its combination of secure end-user data persistence across all nodes, secure pair-wise message exchange between nodes, tamperproof code running on all nodes, synchronous FIFO link communications based on WebSockets, along with secure cryptographic key management, and DDoS resistance, gives UnoSys a technical security *fortress* foundation. This advanced fortified security model makes it possible to create a reliable, state-of-the-art, open, distributed operating system from a collection of inherently untrusted computers, at Internet scale. Moreover, it achieves all of this with no centrally controlled trust authority.

Finally, UnoSys is designed to be a completely autonomous, fully distributed system capable of self-aware analysis in order to police itself – monitoring the parts of itself that are working well and those that are not, and acting accordingly to maintain full reliability at all times.

Beyond its considerable technology, UnoSys also seeks to empower and reward those who contribute resources to its network. It attempts to autonomously motivate end-users to contribute their available spare computer resources to grow the network. Its design encourages end-user behavior in a manner that is aligned with the interests of the commons, while at the same time deterring malicious agents from attacking the system for the purposes of personal exploitation or vandalism. These innovative social and economic design aspects of World Computer Project are explored extensively in the companion document "***A Vision for a New Dignified World Computer Economy***" available on the World Computer Organization website (www.WorldComputer.org).