What is a distributed grid system?

A grid system is a collection of small computers that are designed to all work together as a single big computer, sort of like building a mainframe out of PC's. A distributed grid system adds redundancy on top of that functionality, so that each part of the grid has multiple duplicate locations and copies of data, spread out in separate locations (distributed). The reason for doing this is so that when individual PC's, or even entire locations go offline, the distributed grid system as a whole keeps running nonstop.

What are the advantages of a distributed grid system?

The primary advantages of a distributed grid system are twofold: first, it enables incremental horizontal scalability (scale out) for both processing power and storage capacity in a system; and second, it provides 100% system availability (uptime) due to the redundancy of at least two, and ideally three, distributed locations that are all active and writable at the same time. The incremental horizontal scalability for both processing nodes and storage nodes allows the distributed grid to start as a single computer in each location, and then grow to whatever size is needed just by adding more PC's in each of those locations.

There are other redundant systems available, but they are not able to distribute the data in a system to multiple different locations very well. Their biggest limitation is that only one copy of the data is writable at a time, in order to maintain data consistency. In a distributed grid, on the other hand, all copies of the data are autonomous and writable at the same time, while the data is continuously synchronized between the different locations in the background (eventual consistency). This is what allows a distributed grid system to continue to be available and useable even if multiple locations of the grid are offline.

Who currently uses distributed grid systems?

All of the big Internet-scale companies have taught themselves how to design, build and maintain distributed grid systems, and those grid systems are the foundation for many of their products, services, and even their business as a whole. Distributed grid systems are very complex, but provide huge competitive advantages to the companies that know how to build and maintain them. The Big Tech

companies keep all of their distributed grid knowledge and experience as closely-guarded trade secrets, because the competitive advantages they provide are so good that they act as a "barrier to entry" for other businesses that want to compete with them, but can't since they don't know how to build and maintain their own distributed grid systems for themselves.

What is DGP (Distributed Grid Platform)?

DGP is a simplified distributed grid computing platform that is free and open-source, built using off-the-shelf web servers (Windows IIS) and database servers (SQL Server Express and Standard editions). It is designed to provide the benefits of distributed grid computing systems to SMB's (Small to Mid-sized Businesses). DGP systems must therefore be simple, inexpensive, and able to run well on a single computer – while also being able to satisfy the demand for more processing power and storage as the system grows, with an upper limit that is well beyond the needs of most businesses. In order to meet these requirements, DGP contains several innovations:

- Multi-master merge replication: DGP treats its Replica database tables as append-only collections of immutable records. Merge replication then becomes a process of forming eventually consistent unions of the immutable records created in each location within each of the Replica database tables. "Multi-master" refers to the fact that the databases in each location is an autonomous independent "master" copy of the data, which allows all the copies to be writable at the same time. In addition, the merge replication also contains mechanisms to automatically repair data synchronization after server or location downtime, while the system remains online.

- Scalable database shards: DGP shards provide vertical partitioning between a main database schema and the shard schema(s), as well as the horizontal partitioning of the data in the shards themselves. This allows for the data in the shard tables to be scaled out horizontally as needed in each of the locations of the distributed grid system, while also being fully integrated with the multi-master merge replication processes. The setup of the shards and the process to add new shards is simple to configure, easy to maintain, and very reliable in practice.

- Scalable automated processing: DGP's AutoWork services are a data-driven federated computing system that constantly runs small-batch iterations of all the automated processes, and is able to scale out horizontally to meet increased automated processing workloads by adding new processing nodes to the farm in each location. Automated processes perform all of the merge replication, data consistency checks, in-place data repair, duplicate checks, monitoring queries, etc. All automated work is implemented as web service API methods and are run with the full security of the data-driven RBAC security system.

- End-to-end unit testing: DGP's message-based API's are able to exchange a significant amount of data between the client app and the web service API's running on the server. This capability is used to build a simple form of unit testing into each API method, and communicate the results of those tests on the server back to the client application. This is the foundation for the API Tester test harness built into DGP, which is used to test every API method in a system end-to-end (both positive and negative tests), every time a new build is deployed. In effect, unit tests have been merged into the normal functionality of each web service API method in DGP.

Software Quality

Also of concern to businesses that build their own customized software systems are the multiple problems with current software development practices. CISQ creates a report each year to calculate the cost of these problems for US businesses. The latest version https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf estimates that the total cost of poor software quality for US businesses in 2020 was $2.08 Trillion, not counting the $1.31 Trillion cost of technical debt. These costs have been rising significantly in recent years. The negative effects of poor software quality include:

- Operational software failures due to unmitigated flaws in software systems (excessive downtime, corrupted or lost data, etc.)
- Cybersecurity failures (security breaches, stolen data, destroyed data, etc.)
- Unsuccessful software development projects due to lack of attention to quality (complete or partial failures of projects)
- Flaws in legacy software systems, and the decline in knowledge of legacy system internals

In addition to its distributed grid functionality, DGP has also been designed and built to help address the problems of poor security, poor performance and poor overall software quality.

- Ensure that the functionality of the web service API methods is correct, proven by building unit tests into every API method
- Ensure that the data stored in the locations of a system is correct, proven by verification processes crawling through the data
- Effective software system security, proven by periodic penetration tests, foothold tests attacking the data encryption, etc.
- Good end-to-end performance and comprehensive redundancy that guarantees a system will be usable 7 x 24 x 365
- Delivery of high quality software systems that are proven to meet all requirements, at the lowest cost and in the shortest amount of time possible

The basic premise behind DGP is to correct all of the major system-level software quality problems, and then build software systems using DGP as a foundation so that all of those improvements are shared and reused by the new custom software.

Independently Verifiable Proof of DGP Functionality

One of the most important conventions in DGP is the need for every one of its features and capabilities to be repeatedly tested to *prove* that it is working correctly and performing well. In addition, all of the tests, measurements and data forming these proofs must themselves be verifiable in some way. Several areas of DGP's architecture and code provide the means to test, measure and prove the correctness of all of its functionality every time a new build is deployed:

- Essentially all of the functionality in DGP has been consolidated as web service API's and the reusable libraries of code behind them that do all of the actual work in a system. The data storage tier and the client app tier contain little or no logic.
- The API Tester test harness tests both the functionality and performance of every API method in a system using the end-to-end unit testing feature of DGP. The only exceptions to this rule are some of the automated processes which require multiple locations and configuration data.
- The AutoWork windows service running as a console app is used to test and observe the claiming and execution of automated processes, along with the AutoWorkLog which stores the results of each process iteration when logging is turned on.

The test results that prove DGP's functionality and performance are able to be verified themselves by analyzing DGP's open source code, which includes the built-in test harnesses and the test files they use to run all of the API method tests. On a development computer using localhost endpoints, a full regression of all 600+ API method tests takes between 3 to 4 seconds to complete. Running the same full regression remotely will take longer due to network latency, but will still be a relatively short period of time.

Another similar form of proof of the web service functionality and performance are the client applications themselves. The Lattice application is one example. For all user actions, Lattice calls web service API methods the same way as the API Tester test harness. The application displays the data returned by the various method calls, while the status bar displays the end-to-end time required for the API Request/Response round trip, along with the server-side processing time for the batch of methods in the API request.

These performance measurements constantly prove the performance of the system to all users of production applications under real-world daily workloads, without adversely affecting the production systems themselves. Users that are members of the RemoteMonitor role also save the performance data to the SysMetrics database of the location they are using at the time.

DGP Features, Functionality and Capabilities

The overall purpose of DGP is to provide SMB's with a free and open-source platform for custom software development that has many advanced features and capabilities already built in. Rather than reinventing this necessary functionality for each new system, the reuse of DGP's system-level capabilities improves the security, performance and overall quality of the custom software system. It also reduces the amount of new code that must be written for each system, which in turn reduces both total costs and delivery time. Many of the other capabilities of DGP also help to reduce the total cost and TTM of software development.

Each of the capabilities listed below has design documentation under the Setup section which provides additional information, along with a list of ways that each capability can be repeatedly tested, measured and proven/verified to be working correctly.

- Availability: when using the standard three separate locations, each of which is active and independently writable at the same time, DGP systems are able to provide 100% system uptime 7 x 24 x 365. This level of uptime depends on DGP's multi-master

merge replication and automation subsystem to continuously synchronize the data between each of the three locations. Refer to the Availability documentation under the Setup section for more information.

- Scalability: the requirements for scalability on the low end is that all the tiers of a DGP system can be easily installed on a single computer, for software development, fast debugging, and small-scale production systems. The basic premise is that most systems should start out at the smallest scale possible and only grow if and when it is needed. The requirements on the upper end is that a multi-tiered DGP system can be scaled incrementally to a size whose upper limit is beyond the needs of almost all businesses. Refer to the Scalability documentation under the Setup section for more information.

- Automation: DGP's automated processing is handled by a federated computing subsystem that continuously runs processes for data replication, error detection, in-place data repair and monitoring in each location. All of these processes have been implemented as web service API methods, which allows them to take advantage of the performance and scalability of the load balanced web server farms, and are executed using the full security of the data-driven RBAC subsystem. Refer to the Automation documentation under the Setup section for more information.

- Security: DGP has the equivalent of an API security gateway built into each web service in the form of a message processing pipeline for the message-based API's, which uses the data-driven Rule Based Access Control (RBAC) subsystem to provide fine-grained control over the authorization of each individual API method. The standard for DGP security is to be able to pass a PCI-DSS audit. That standard represents a very high level of effective security, even if no audit is ever performed. In addition, there are no default admin accounts, passwords or even naming conventions in a DGP system (all of which are customized during setup). Finally, DGP only uses the closed-source .NET Framework 4.8, and has no dependencies to any 3rd party tools or libraries. This eliminates the need to scan the tree of open-source software and all of its dependencies for malicious code. Refer to the Security documentation under the Setup section and the message pipeline section under Web Services for more information.

- Performance: DGP uses performance engineering techniques to deliver native executables that are generally 10 to 20 times faster than typical systems built using OO standard practices. The performance requirements specify that every user action in a client UI must be displayed and the UI fully interactive in less than one second, end-to-end. To meet that requirement, every API method

must be completed on the server in less than 100 MS (on average the server-side processing of DGP API methods will be completed in less than 10 MS, with full security). Refer to the Performance documentation under the Setup section for more information.

- Testing: DGP contains a built-in test harness because testing is so important in DGP systems that it cannot be left to chance. The API Tester test harness tests the functionality and end-to-end performance of almost all of the API methods in a DGP system (both positive tests and negative tests combine for 100% code coverage of each method tested). The 260 API methods in the beta system can be tested by the API Tester application in between 3 to 4 seconds (localhost, no network latency). In addition, several automated processes are used to test and verify the correctness of the data stored in each location of a system. Refer to the Testing documentation under the Setup section and the Testing section of the Lattice documentation for more information.

- Logging: Logging is a process of reactively storing data about events that occur in a system. DGP is built using both distributed and centralized logging mechanisms for all of its tiers. All exceptions, errors and event info are first logged to the local event viewer of each computer, which is guaranteed to work in almost all circumstances. A second step then stores the same info to a centralized database in each location by calling an API method if running remotely, or by direct connectivity to the database server when running on the internal network. Refer to the Logging documentation under the Setup section for more information.

- Monitoring: Monitoring is a process of proactively collecting data about a system as it runs, checking to insure that it is functioning correctly and performing well. One example of the monitoring built into DGP systems is the Lattice application. Users that are members of a special monitoring role save the status bar performance metrics of the app to the SysMetrics database of the location they are connected to during their use of production systems. This measures the functionality and performance of the system under realistic production workloads at all times of the day, without adversely affecting the system being monitored. Refer to the Monitoring documentation under the Setup section for more information.

- CI-CD-CT: The architecture and code of DGP systems have been designed to support CI-CD-CT processes. Automated testing of all the functionality of a system is a prerequisite for CI-CD-CT, which is both simple and easy using the test harness applications built into DGP systems. DGP uses the .NET XCOPY option for the deployment of all of the tiers of a DGP system as folders, using an

immutable append-only process which greatly simplifies the deployment and rollback of new versions. In addition, the DBSetup utility is used to maintain the database schemas and core data in each environment and location of a distributed DGP system. Refer to the CI-CD-CT documentation under the Setup section for more information.

- Evolution: The architecture and code of DGP systems have been designed to support service and system evolution, which are the sum total of the many extensions, enhancements and improvements added to a custom software system over time – all without ever breaking backward compatibility. This capability depends on the abstraction of the message-based API's from the internal libraries of code that do all of the actual work, used in conjunction with immutable append-only conventions followed for API methods, versions and database schemas of a DGP system. Refer to the Evolution documentation under the Setup section for more information.

- Compatibility: The objective for universal compatibility is for a single instance of DGP web service API's to be able to be used by client applications written in any of the major programming languages, without the need for client SDK libraries. The primary purpose of this compatibility is to significantly reduce the time and cost of supporting and maintaining a DGP system over its productive life. Refer to the Compatibility documentation under the Setup section for more information.

DGP Lattice

Lattice is a free, open-source application which has been built as a distributed grid system using DGP as its foundation. The Lattice UI functionality provides admin tools to manage security, configuration, testing, and monitoring, along with file storage functionality similar to a sharable version of OneDrive or DropBox. Its open source code also acts as a reference implementation that provides many working examples of how to use the different types of DGP functionality in practice.

- Connect Section: Each distributed location is active and writable at the same time, so users must select which location they wish to connect to with the help of a system list file. Users only connect to one location at a time, which gives them full sequential

read-after-write consistency for their own data. All data created or edited by users is automatically replicated to the other locations by AutoWork processes running continuously in the background (the eventual consistency is hidden from most users).

- Security Section: The admin UI's manage the data-driven Role Based Access Control (RBAC) security system used to authenticate user accounts and individually authorize access for every API method in a DGP system. Data groups are used to partition shared data in the FileStore application to control which accounts have access to which data groups.

- Configuration Section: The admin UI's manage the data-driven configuration of the AutoWork processing system used to continuously run automated processes in the background, guaranteeing the "once and only once" execution of each task:
  - Multi-master merge replication processes
  - Distributed data verification processes
  - In-place data repair processes
  - System monitoring processes

- Testing Section: The admin UI's of the two built-in test harness applications:
  - API Tester: Runs sets of both positive and negative tests for almost all API methods in a system, which is made possible by the functionality of the message-based API's plus the use of custom API method test files and test harness logic.
  - Log file viewers: Screens to view and search DGPErrors, LatticeMetrics, AutoWorkLog and TestResult tables (paginated).

- User Section: Data forms for each user to manage their own profile data, update their password, and so on.

- FileStore Section: File storage and collaboration similar to a sharable version of OneDrive or DropBox
  - FileStore can be used to securely store any type of file in its distributed storage. File segments are stored as Base64 encoded text in each segment record of a shard database.
  - A simple shard mechanism allows the data storage shards to scale out horizontally as needed in order to store very large amounts of data content. As each shard server is filled with file segment records, one or more new shard servers are added to the collection of shards in each location.

- Each location in a Lattice system is writable at the same time as all the other locations. Content files and metadata records are constantly synchronized between the distributed locations by the DGP merge replication processes running in the background, which allows 100% uptime for the system as a whole in spite of individual servers (or entire locations) periodically going offline.
- All files stored in Lattice are linked to one or more data groups, and membership in those groups:
  - Restricts which folders and files are visible to each user of the system.
  - Controls which users have read only vs. read/write access to the folders and files they have been authorized to use.
- Files can be found by browsing the folder directory structure, by searching based on file metadata, by searching for files linked to tag values, or by saving links to frequently used files in the user's Favorites folder.
- Lattice stores and preserves all versions of the files in each system, as well as which user created each version, and when. This allows all changes to be rolled back to previous edited/archived versions at any time by Lattice users (including the recovery of deleted files).

Refer to the Lattice documentation under the Client Tier section for more detailed information regarding Lattice functionality.

Reducing the Total Costs of Software Development

In practice, reducing the total costs of software development and maintenance is accomplished by designing and building a system so that it needs and uses less resources of all types (innovation), and/or substitutes the use of less expensive resources to replace more expensive resources. Those are the only 2 realistic ways to reduce costs, and DGP makes extensive use of both of them.

First, performance engineering practices are used to deliver very fast and efficient .NET executables. These performance improvements are mainly accomplished by simplifying both the system architecture and code, which in turn helps to significantly reduce the amount of code needed to implement all of the desired functionality.

The simplified architecture and code, along with other performance engineering techniques, combine to deliver .NET executables with performance that is somewhere between 10 and 20 times better than systems built using standard OO best practices.

Cost Reduction (find ways to use less resources, switch to less-expensive resources):

- The simplified architecture and small amount of simple code reduce the number of developers and testers needed to build and maintain a system. Also, the simple code means that less expensive junior level developers can easily maintain the system.
- Following immutable append-only conventions means that small teams of developers and testers are all that are ever needed to build and maintain large systems, since they only need to handle the net-new development and testing. There is little or no maintenance required for the immutable API methods and test files once they are in production.
- The high performance executables act to reduce the amount of all hardware, software and infrastructure needed to support a given number of concurrent users. This also reduces the number of technical staff needed to maintain the small infrastructures.

Reducing the Time-To-Market of Software Development

TTM is reduced by doing less work to create deliverables (innovation), significantly reducing the amount of maintenance work in systems by following immutable append-only conventions, and by automating as much of the simple work as possible. These reductions can be proven by analyzing the data in the tools used to manage software development (Rally, Jira, etc.).

Time-To-Market Reduction (find ways to do less work, and automating simple work):

- Building new software systems using DGP as a foundation significantly reduces the amount of new code that must be written.
- It takes much less time to implement new functionality in a DGP system due to the simplicity of the overall architecture, its small and simple code base, the high degree of code reuse and the need to only write code to implement net-new functionality.
- The immutable append-only conventions eliminate maintenance work on all existing code, other than occasional bug fixes.
- Automated testing using the built in test harnesses can run full regression tests of all API methods in a system in a few seconds.
- Easy deployments, testing and rollbacks means that these processes are also quick and easy to perform manually or to automate.