Architecture and Design

Version 1.0
July 2006

Contents

Contents.....2
Overview.....3
Copyright and trademark information.....3
Feedback.....3
Acknowledgements.....3
Modifications and Updates.....3
High-Level Goals.....4
Development Practices and Principles.....6
The Directory Server Components.....9
The Server Configuration.....12
Client Connection Handling.....14
The Backend Databases.....16
Synchronization and Change Notification Mechanisms.....19
Access Control Rules.....22
Other Security Features.....24
Logging and Alerting Mechanisms.....28
The Plugin API.....31
Supported Controls and Extended Operations.....34
Common Development and Distribution License, Version 1.0.....36

Overview

This document describes the overall architecture and design of the OpenDS Directory Server.

Copyright and trademark information

The contents of this document are subject to the terms of the Common Development and Distribution License, Version 1.0 only (the "License"). You may not use this document except in compliance with the License.

You can obtain a copy of the License at https://OpenDS.dev.java.net/OpenDS.LICENSE or at the end of this document. See the License for the specific language governing permissions and limitations under the License.

Portions created by Sun Microsystems, Inc. are Copyright © 2006. All Rights Reserved.

All trademarks within this document belong to legitimate owners.

Feedback

Please direct any comments or suggestions about this document to:
issues@opends.dev.java.net

Acknowledgements

The general format of this document was based on the documentation template used by OpenOffice.org.

Modifications and Updates

Date	***Description of Change***
07/19/06	Initial version.

High-Level Goals

The primary goals for the OpenDS Directory Server include:

- **Performance**. It must be extremely fast, outperforming all other servers on the market wherever possible.

- **Upward Vertical Scalability**. It must be capable of handling billions of entries in a single instance on adequately-sized hardware. It should be able to make effective use of hundreds of gigabytes of memory or more, and it should be able to make effective use of at least 32 CPU cores.

- **Downward Vertical Scalability**. It must be capable of running adequately in low-memory environments so that all essential components may be functional with a Java heap of no more than 16 megabytes.

- **Horizontal Scalability**. It must be possible to use synchronization to achieve horizontal read scalability. It must be possible to use data distribution in conjunction with synchronization to achieve horizontal read and write scalability.

- **Supportability**. The server should work properly and in an intuitive manner wherever possible. If a problem does arise, the server must be able to provide sufficient information to determine the cause of the problem so that it can be corrected without as quickly as possible. All messages that may be written to the error log or included in a response to the client must be given unique integer IDs. The server must support common monitoring frameworks, and must provide the ability to generate alert notifications in the event that a significant problem is detected.

- **Security**. The server must provide unmatched security in areas like access control, encryption, authentication, auditing, and password and account management.

- **Extensibility**. Virtually every aspect of the server should be customizable. It will have a safe and simple plugin API (with more plugin points than the current Sun Java System Directory Server), but will also have additional points of extensibility, including password validation algorithms, password generators, monitor information providers, logging subsystems, backend repositories, protocol handlers, administrative tasks, SASL mechanisms, extended operations, attribute syntaxes, and matching rules.

- **Synchronization**. The server must support data synchronization between instances, including not only total data synchronization but also partial synchronization (with fractional, filtered, and subtree capabilities), and must also provide a means of synchronizing with other applications.

- **Availability**. The server must be robust so that it can continue running properly even in the case of all but the most severe errors. Virtually all configuration changes should be able to take effect without restarting or significantly impacting availability. The server should have a mechanism for attempting to notify proxy servers and/or load balancers if a change will be made to its availability so that future requests can be routed away from that server if possible.

- **Portability**. The server must be written entirely in Java so that it may run on any platform with a Java SE 5.0 or higher runtime environment. Any administrative and/or wrapper scripts needed to invoke the Java code should be written as portable Bourne shell scripts for all UNIX (and UNIX-like) platforms, and batch files for Windows systems. Platform-specific elements must be kept to a minimum, particularly those requiring native code, and must not be required for running the server. If binary components are absolutely required (e.g., to allow the server to run as a Windows service), then the compiled binary component must be checked into the repository so that the entire server can be built and packaged on any platform.

- **Migratability**. Even though the Sun Java System Directory Server will continue to be maintained for some period of time and will not be immediately supplanted by the OpenDS Directory Server, the OpenDS Directory Server server must provide support for virtually all existing features of the Sun Java System Directory Server. Any functionality that is not being carried forward should be explicitly identified and justification provided to indicate why it has been deprecated. Note that this requirement does not mandate that the feature is implemented in the same way or using the same syntax if the change offers a distinct advantage, although the migration path should always be taken into account in such cases.

  Migration from other directory server implementations should also be taken into consideration when applicable. Automating the migration process in these cases may be a lower priority, but we should attempt to identify those features in other servers that may make migration difficult so that we can include support for them when possible.

Ideally, the server will also meet the following criteria:

- It will have a clean code base that is thoroughly commented.

- It will offer good standards compliance and will provide support for proposed standards in areas where there is demand. This will include supporting a wide array of controls and extended operations.

- It must provide perfect compatibility for applications using purely standard LDAP operations with no implementation-specific focus. It must also offer perfect compatibility with some of the most widespread LDAP-enabled applications.

Development Practices and Principles

One of the most significant design decisions to make for the Directory Server is the programming language to use when developing it. Several existing servers are written in C, which has a number of advantages like potential for high performance and significant expertise by existing engineers. However, it also has a number of notable disadvantages, including poor error handling, lack of object-oriented design, and complicated portability issues. C++ improves upon error handling and object-oriented capabilities, but is less standardized than C and has more portability issues.

The ideal development language to use for the Directory Server is Java. It excels at portability and provides superb error handling and object-oriented development. It also offers great support for threading and networking, and includes a wide array of data structures and libraries included as part of the standard runtime environment, and includes capabilities for developing graphical and Web-based interfaces. Further, programming in Java encourages good coding practices because the Javadoc convention makes it easy to generate basic documentation from the source code, and Java code is inherently more readable than that from other languages like C or C++.

It is a commonly-held belief that Java has poor performance. While that may have been true in early releases, in general it is possible to write Java applications that are as fast or faster than the same application written in C. One big benefit that Java can provide in this area is the ability of the just-in-time compiler to dynamically optimize the code based on profiling, instead of creating static binaries based on the compiler's best guess about how the code will execute. This benefit is particularly observable in long-running applications (like a Directory Server) because the JIT compiler is able to collect more data to make its decisions. Further, tests run previously with a Java-based LDAP Distribution Server and with very early builds of a Java-based Directory Server yield extremely impressive performance, several times better than can be achieved using an existing C-based server on the same system.

As Java is to be used for developing the Directory Server, the next important issue to resolve is which runtime version should be targeted. Because the server must be able to handle large numbers of concurrent connections that may or may not be active at any given time, the use of NIO is essential, and therefore Java 1.4 or later is required. However, the implementation provided in Java 1.4 does not include a mechanism for communicating over SSL/TLS when using NIO, and therefore it will be necessary to use Java 5.0. There are additional benefits to targeting the Java 5.0 environment, including:

- It includes a number of performance enhancements over Java 1.4. New concurrency libraries make it easier to develop more scalable applications, and garbage collection improvements and additional compiler optimizations can generate faster, more highly optimized code.

- Support for generic data types makes it possible to perform more strict checks at compile-time that will prevent potential runtime errors.
- New language conveniences like the enhanced for loop, varargs, and static import can reduce development time.
- The Java Management Extensions (JMX) library is included as part of the Java 5.0 environment, which can provide hooks into management and monitoring systems.
- It is the latest official release, which means that it will be supported farther into the future than older releases.
- It is the first release to provide 64-bit support for x86-64 systems (AMD64 and EM64T).

In order to provide the greatest degree of stability and portability, all components of the server should be pure Java. The use of native code should be avoided at all costs whenever possible. This applies to both code within the core server (e.g., through the use of JNI) and mechanisms used to install and manage the server. Shell scripts and/or batch files may be exempt from this in some cases because they do not require compilation and therefore would not prohibit the process of building the Directory Server on one platform and executing it on another. If there is any absolute requirement for native code for certain modules, then it is recommended that they be built and distributed separately so that the core Directory Server code can be built once on one platform and executed on any platform with the appropriate Java runtime environment (even those platforms that are not actively supported for production use, but users may find useful for testing purposes).

All source code must be fully documented using the Javadoc style, and the Javadoc generation process must complete successfully with no warnings or errors. This documentation should include a summary of each class or interface, a summary of each method (including all arguments used and any exceptions that might be thrown), and a summary of each field that is accessible outside that class. Whenever appropriate, instance variables defined within a class should be declared private and any access to or manipulation of such variables must only be allowed through methods. Support for generic data types must also be used wherever possible to both simplify the code and reduce the chance of runtime errors due to inappropriate type conversions.

Dependence on external code must be kept to a minimum. With the possible exception of certain complex elements like the backend database in which performance and reliability are critical, all code should be developed within the project, making use of only data structures and methods provided by the J2SE 5.0 API. This dramatically increases the portability and supportability of the Directory Server, reduces legal concerns from the use of third-party code, and eliminates concerns about the viability of external components. Note that some third-party code may be necessary in some cases (e.g., a JDBC driver for interacting with an external RDBMS), but in general they should be fringe modules and not within the core of the server. If necessary, additional standardized Java APIs may also be used to extend non-core functionality (e.g., the use of J2EE may be needed if it is desired to add an HTML-based administration interface).

The Directory Server must also be designed with supportability in mind. Any LDAP result returned to a client containing a result code other than success (0), compareFalse (5), compareTrue (6), or referral (10) should include a clear and detailed errorMessage string providing additional information about the underlying problem. Any client connection closure that is not initiated by the client or that is the result of a communication problem must be accompanied by a notice of disconnection extended response that describes the reason for the closure. The server must also provide clear and detailed log messages for any problems or notable events that occur during processing. All messages that may be written to the server log or that may be sent to the client in an LDAP response must be assigned a unique number and must support localization.

Debug messages may be embedded within the code in order to help the support and/or engineering organizations more effectively diagnose problems that cannot be solved through log messages or LDAP responses. This debugging capability should be included within all development and stable builds of the Directory Server in order to avoid the need to generate and test multiple builds. The Java assertion mechanism should be used in all such cases, however, so that debugging is only possible if assertions have been enabled within the virtual machine. With assertions disabled, the debugging statements should have no impact on the performance or operation of the server. If the need arises, assertions may be enabled globally or on a per-package or per-class basis to make it possible to perform this additional debugging. Note that all use of assertions in the Directory Server code must be guaranteed to return a `boolean` value of `true` so that there is no chance that a failed assertion will interfere with the processing of the server.

In the past when Java has been used as a development platform, some have expressed concern that Java bytecode may be too easily de-compiled into source form. This is irrelevant in an open source development project. It is possible to use obfuscation tools in order to inhibit decompilation, and while they may also offer additional benefits (e.g., slightly better load times due to smaller classes) they can make the resulting code much more difficult to support and debug.

The Directory Server Components

The Directory Server should be constructed using a very modular architecture in which most or all components are written to a well-defined specification so that alternate implementations may be created and installed instead of or in conjunction with the default set of components. Some of this extensibility may be exposed to users so that they may add their own custom logic into the request handling process. Other interfaces may not be intended for users but may be available for limited use by developers with more specialized knowledge of the server architecture. Still other interfaces may be intended for project use only to preserve the modularity and extensibility of the server.

The set of components that will comprise the Directory Server includes:

- Connection and request handlers that manage all interaction with clients. This includes accepting new connections, reading requests and ensuring that they are handled properly, and returning responses to those clients. These modules will be responsible for any special processing that may be required for this communication, including managing encryption or performing protocol translation. It must be possible to have multiple concurrent implementations active at any given time in order to facilitate all forms of communication that clients may wish to use to interact with the server, and it must be possible to enable or disable them on the fly as necessary.

- A work queue that will be responsible for ensuring that all operations are processed in an efficient manner. It will likely employ several worker threads able to process requests concurrently, and it will also likely include a queueing mechanism that can be used to temporarily hold requests that have been received until a worker thread can process it.

- A configuration handler that is responsible for managing the storage and retrieval of configuration information, as well as providing notification to all appropriate components whenever a configuration change occurs. Only a single configuration handler should be active at any given time and it should not be possible to replace or disable the configuration handler while the server is running.

- A set of backend databases that are used to search, retrieve, and store the directory data. There may be multiple backend databases (and database implementations) active at any given time within the server, each of which handle mutually exclusive subsets of the data. It should be possible to enable, disable, create, remove, backup, and restore databases independently of each other and without impacting the availability or operation of other databases. It should also be noted that some backends may not use local repositories, but rather access data stored remotely (e.g., as in a proxy or virtual backend), or in some cases may generate the data on the fly (e.g., as in a monitor backend).

- A set of access loggers that can be used to record information about the operations processed within the Directory Server. Multiple access loggers may be installed and active at any given

time, where each may log to a different repository, present the information in different formats, or implement filtering so that only certain types of operations are recorded.

- A set of error loggers that can be used to record information about warnings, errors, and significant events that occur within the Directory Server. Multiple error loggers may be installed and active at any given time, and each may be configured independently. Further, some error loggers may be used as an alerting mechanism to actively notify administrators of potential problems in the directory environment.

- A set of debug loggers that can be used to record debugging information that may be generated if the server is run with debugging enabled and Java assertions active. Although it is unlikely that there will be a need for multiple concurrent debug loggers active within the server at any time, it should still be possible to do so if the need arises.

- An access control module that is used to determine whether or not a given request should be processed, and if so which information should be returned to the client. Only a single access control module may be defined within the server and it may not be replaced while the server is online. However, the access control rules that are used to control what operations users are allowed to perform may be altered on the fly.

- An entry cache module that is used to store entries in memory for faster access than may be achieved by retrieving them from the database. Only a single entry cache should be in use at any given time, and it should be possible to disable the cache entirely for cases in which it is not needed. The server should strive to provide adequate performance without the need for an entry cache so that it will be able to make more efficient use of available memory resources.

- A set of synchronization modules that may be used to ensure that changes in the directory configuration or data are propagated to all appropriate servers, and optionally to other systems outside of the immediate directory environment. Synchronization modules may also be used to ensure atomicity for certain operations in the directory environment, including ensuring global uniqueness and consistent account status information. Multiple synchronization modules may be installed and active at any given time.

- A set of handlers for SASL mechanisms supported for bind operations within the server. Multiple such modules may be installed and active at any given time, and each module may implement support for one or more SASL mechanisms. However, each SASL mechanism may be handled by at most one module.

- A set of handlers for processing special types of extended operations within the server. Multiple such modules may be installed and active at any given time, and each module may implement support for one or more extended operations. However, each type of extended operation may be handled by at most one module.

- A set of user-definable plugins that may be invoked at various points within the connection handling and request processing logic. These plugins may perform any number of different kinds of tasks, including dynamically re-writing portions of request and response messages,

providing alternative implementations for certain operations, or any other additional processing that may be required as part of request handling. Multiple plugins may be installed and active within the server at any given time, multiple plugins of the same type may be registered concurrently, and a single plugin may be registered multiple times to be invoked at different stages of request processing.

- A set of syntaxes and matching rules that define the logic for dealing with different kinds of attributes. Multiple syntax and matching rule modules may be installed and configured concurrently, but each attribute syntax must be implemented as a separate module, and each matching rule must be implemented as a separate module.

- A set of modules that define password storage schemes that may be used to obscure user passwords using a reversible or one-way algorithm. Multiple password storage scheme modules may be installed and active concurrently, and each module may register itself to handle multiple schemes.

- A set of modules that define logic used to determine whether a proposed password is acceptable for a user. Multiple password acceptability modules may be installed and active concurrently within the server.

- A set of modules that may be used to provide notification to users for certain events that occur in the account life cycle (e.g., the password is about to expire, the account has been locked, the password has been reset, etc.).

- A set of modules that may be used for identity mapping purposes, in order to correlate a user from a SASL bind request (e.g., based on a username or client certificate) or a proxied authorization control to the appropriate user entry in the directory.

- A set of modules that may be used for performance and/or availability monitoring. Multiple such modules may be installed and active concurrently within the server concurrently.

- A set of modules that may be used to define virtual attributes or attribute values within the data. Multiple such modules may be installed and active within the server at any given time, and each module may be used for multiple attributes. However, each virtual attribute must be managed by only a single module.

Other component types may be defined as needed. Great care should be taken to ensure that once the API for a component type has been defined, and particularly after it has been used in a stable released version of the server, that it remain consistent so that backward compatibility is ensured between releases. This should be especially true for APIs that are intended for external use so that users and other third parties are not required to alter their code between releases (although it may be beneficial to do so in order to utilize new features). To ensure this, APIs defining the various components of the server should be based on abstract classes rather than interfaces whenever possible. Interfaces should be used only in cases in which the lack of multiple inheritance make implementation based on abstract classes infeasible.

The Server Configuration

One of the most important components of the Directory Server is the one responsible for managing the server configuration. This module is responsible for ensuring that all other components have the information that they need in order to complete their initialization and to function properly. It must also validate configuration changes from the end user and provide notification to the appropriate components whenever a change does occur.

Another interesting capability that the configuration manager should provide is the ability to centralize some portion of the configuration so that it is consistent across the directory environment. For example, if an administrator makes a configuration change to enable password expiration every 90 days, then it may be desirable to have that propagate to all servers. However, if an administrator wanted to change the IP address on which the server listened for client connections, then that would only apply to the local system. It should also be possible to administer a Directory Server in a completely standalone mode without synchronizing any data or configuration changes (and this should be the default behavior).

In many respects, the configuration manager should appear as a normal backend database. The configuration information should be able to be read and updated over protocol (subject to access controls, and potentially only for certain protocols). However, it should also have a couple of additional important qualities, including:

- Components will be able to register themselves to be notified of changes to specific configuration entries so that they can determine whether a requested change is valid and if so react to it. If a detected change is invalid, then the registered component will be able to reject the change and provide a human-readable message describing the problem.

- Components will be able to register themselves to be notified whenever a new entry is added or an existing entry is removed below a given parent entry so that they can validate and/or react to that change. If a detected addition or deletion is invalid, then the registered component will be able to reject the change and provide a human-readable message describing the problem.

- Whenever feasible to do so, configuration changes should take effect immediately without requiring a restart of the Directory Server or any further administrative interaction. In some cases, it may be acceptable to apply this only to future events (e.g., any new connections established or operations requested). In the event that it is not feasible to apply a change immediately, then it should be determined whether it may be feasible to implement through some additional administrative action less severe than stopping the server (e.g., by briefly placing all or part of the server in a read-only mode). Changes that require a restart to take effect should be avoided if at all possible.

- The configuration handler should provide a mechanism that can be used to determine whether a particular change will take effect immediately or will only become active after further administrative interaction. In the event that further administrative action is required, it should

be possible to obtain a human-readable string indicating the action necessary to apply the change.

- If a change is made to a configuration attribute and that change cannot take effect immediately, then it must be possible to determine both the currently active value(s) for that parameter and the value(s) that will be in effect after the appropriate administrative action has been taken to apply that change.

- It should be possible to provide alternate means to access and manage the configuration in addition to the supported protocols within the server. For example, it may be useful to provide a Web application that may be used to interact with the configuration in addition to making changes over LDAP. It may also be useful to provide access to the configuration via JMX. Presumably other kinds of configuration interfaces might simply be frontends that communicate over an existing protocol (like LDAP) to actually make the configuration changes.

- It should be possible to perform configuration archiving so that configuration changes can be tracked. If file-based configuration is used and those files were modified with the server offline, then the only action possible may be to compare the file contents against a digest of the last known good file and if a difference is detected then indicate the time that the change was detected. For changes made over protocol or through some authenticated interface, the identity of the user that made the change should be included as well.

The configuration management capabilities of the OpenDS server will not be compatible with any existing Directory Server implementation. In particular, because the OpenDS server will support new features and will have different implementations for many existing features, it does not make sense to require full backward compatibility. Further, even in cases where backward compatibility may be possible it may not be desirable to do so because most existing configuration attributes are prefixed with product or vendor names. To avoid this type of problem in the future, the names of all configuration attributes should not include the project name.

Client Connection Handling

Another significant portion of the Directory Server lies in the way that it interacts with clients. It is the job of the connection handler to accept new connections from clients and ensure that any requests that they submit are properly handled. This includes parsing the requests from the client and translating them into a form that can be understood by the core server. It also includes the logic to translate the response from the core server back into a form that the client can understand. In this way, it should be very straightforward to add support for additional protocols by adding new connection handler implementations. The initial implementation, however, will only include support for LDAP and LDAPS and other protocols may be evaluated and added in the future.

Note that in this context, the term "client connection handling" does not necessarily preclude the "client" being an application in which the Directory Server is embedded. In this case, the communication with the "client" would be in the form of direct method calls rather than actual network communication. In such a case, it may be desirable to create a user-friendly SDK for wrapping the requests in a form that the Directory Server can parse. If this is deemed an important use case for the Directory Server, then it should be possible to operate the server in a mode that does not require any network interaction at all.

In order to ensure that the connection handling is as efficient and scalable as possible, it will use the NIO library to use a relatively small number of threads to handle communication from a potentially large number of clients. It will have the following features:

- The ability to add, remove, enable, and disable connection handlers on the fly without the need to restart the server.
- The ability to enable or disable connection handlers without impacting connections that may have already been established (i.e., don't accept any new connections but don't disconnect those that are already established).
- The ability for an administrator to forcefully disconnect a particular client for some reason.
- The ability to consult allowed and/or denied client lists when deciding whether or not to accept a connection from a client.
- The ability to consult a CRL or OCSP service when determining whether to trust an SSL-based connection in which the client presents its own certificate to the server.
- The ability to use the LDAP notice of disconnection extended operation whenever the server initiates the process of closing the connection to a client for some reason (e.g., idle timelimit, protocol error, server shutdown, etc.).

Preliminary testing has shown that there can be significant improvement by balancing the decoding process across multiple threads (e.g., by alternately registering incoming connections with different selectors), rather than using a single thread. All reads will use non-blocking operations so that clients communicating over slower networks (or those that are intentionally slow in attempt to create a denial of service) will not impact operations on other connections. Further, the thread used to read a request from the client will not perform any processing on that request for any type of operation other than unbind and abandon (as they can be processed very efficiently and do not require any response from the server). All other requests read by the server will be placed in the work queue so that the more expensive processing can be done in parallel.

The Backend Databases

In order for the server to be of any real use in storing and retrieving data, it must have one or more repositories to hold that information. These repositories are the backend databases, and they will be based on an extensible framework so that ultimately it may be possible to configure the server to use a variety of repositories for actually storing the data (e.g., embedded databases, external databases, flat files, or other network services), or even to provide more complicated functionality like might be available in a virtual directory or LDAP distribution server. It could even be possible for a backend to have no actual data but rather dynamically construct entries upon request based on some set of criteria. It must be possible to have multiple backends of the same and/or different types configured and active within the server at the same time, and it must be possible to perform administrative and maintenance functions one one backend without impacting the others. Further, if an administrative or maintenance operation does require that a backend be unavailable for some period of time, then it must be possible to temporarily re-register the affected portions of the directory hierarchy with some other mechanism (e.g., one that uses chaining or referrals to dynamically re-route requests for that data to another server in which that data is still available).

The particular backend or set of backends to use when handling a request should be determined based on the targeted set of data. Each backend must register the suffix or set of suffixes that it handles on startup. A single backend may be responsible for multiple suffixes, and the sets of registered suffixes must be mutually exclusive (i.e., two different backend types must not be allowed to claim responsibility for the same suffix). Sub-suffixes will be supported, but it is recommended that they only be used in cases in which a different backend type is required for a portion of the content (e.g., the majority of data is local, but one branch references a remote repository). The use of sub-suffixes introduces significant complexity in the environment because it can be difficult to maintain consistency among results from multiple repositories when the request would require special processing like sorting or retrieving a specified subset of the results.

The root DSE of the directory server should be treated as a special case. Its content will be entirely dynamically-generated and therefore it will not be part of any backend. However, for onelevel or subtree searches in which the base DN is the root DSE, it will be seen as the parent to all other backends in the directory. This will include the server configuration but may not include other dynamically-generated entries like the schema definitions or entries used for monitoring performance and/or availability.

The interface to a backend must provide a number of capabilities:

- The ability to retrieve an entry based on its DN.
- The ability to add a new entry.
- The ability to remove an existing entry.
- The ability to replace an existing entry.
- The ability to process searches, optionally sorting and/or retrieving only a subset of the results.
- The ability to perform "binary" backup and restore capabilities.
- The ability to perform import from and export to LDIF.

These operations are relatively generic, but this is intentional because it allows most of the processing to be performed in the core server code rather than code that is specific to the backend. This makes it much easier to add additional backend types if necessary while keeping equivalent functionality and further encouraging the modular nature of the Directory Server.

Although extensibility will be a large focus of the backend and the Directory Server in general, there will be one primary backend type that will be used for most or all installations. It will have the following characteristics:

- It must be an embedded database. It cannot be a separate process running locally or on a remote system as this would likely create performance, security, and management problems.

- It must be possible to access the same instance of the database concurrently by multiple separate processes on the same system. One of those processes must be able to have write access to the database, while all others must be allowed read-only access. This capability may be required by certain backup and/or analysis tools.

- It must be capable of delivering extremely high performance. Ultimately it should be possible to achieve rates of tens of thousands of exact searches per second and thousands of updates per second.

- It must be highly scalable so that it can potentially support several billion entries (i.e., the number of entries must not be constrained by a 32-bit integer).

- It must be a transactional database that can provide full ACID (atomicity, consistency, isolation, and durability) assurance. Ideally, it should also be possible to configure the database to reduce strict compliance in one or more of these areas for improved performance and/or scalability if the administrator understands and is willing to accept any associated risks.

- It should use a B-tree or similar mechanism for storing and accessing data. It should not use a relational model because they are generally less efficient at performing directory-related operations.

- It must be pure Java, and therefore should not require native code or use JNI interfaces. If the core database for the server were to use any native code, it would become a hindrance to portability, and could also adversely impact stability because any bug or fault in native code would likely crash the JVM and therefore the entire Directory Server. If it is pure Java, then there should be no portability concerns, and problems can be handled through the exception management capabilities.

- If the database provides a caching mechanism, it must be possible to ensure that this cache will be aware of and can deal with constraints placed on the memory available to the JVM.

- It must have a relatively stable interface that should avoid required code changes in order to update to a new version of the database. It may be desirable to make code changes in order to take advantage of new features, but it should not be necessary to make any changes in order to benefit from the potential fixes and/or performance enhancements offered by the newer version.

- If the on-disk format for the underlying data files changes between database versions, then new versions must be able to read earlier formats, and either write using the older format or perform an on-the-fly upgrade that will not significantly impact the availability of the information during that process.

The preferred backend type will be the Berkeley DB Java Edition. The standard Berkeley DB does provide a Java API, but it uses JNI to invoke native code which is highly undesirable. Many other popular alternatives that could potentially meet the scalability requirements use a relational database model, which is also undesirable. Preliminary testing has shown that the Berkeley DB Java Edition should meet all the above requirements.

The format in which the information is stored in the database should use a custom binary representation that is optimized for very fast access and conversion into a form that may be used by the core server. It may be desirable in some cases to make trade-offs in order to achieve this performance (e.g., storing data in both the user-provided and normalized forms for faster comparisons), but this should be investigated more carefully because it would have the potential to significantly increase the database size (which may in turn have the potential to adversely impact performance, and would certainly increase backup times). It is believed that it will have notably better performance in some cases, in particular for entries like large static groups that may have a large number of attribute values using the distinguished name syntax. If an optimization like this is included, it may also be desirable to make it configurable to allow the administrator to decide whether or not to perform such expansion so that it can be avoided in environments where the improved performance is not required and does not justify the increased storage requirements.

Another property that should be investigated is whether or not complete distinguished names should be stored in their complete form in the backend database, or whether they should be dynamically constructed whenever they are required. Constructing DNs dynamically could make certain kinds of operations (in particular, modify DN operations on non-leaf entries) more efficient, but they could come with a significant cost as well. In particular, it would likely adversely impact run-time performance and could significantly increase the complexity of storing information for attributes with a distinguished name syntax. This could be especially true for attribute values with a DN syntax that do not refer to an entry that actually exists in that database (either because they have not been created or because they are contained in another backend or some other remote repository). Because of the complexity involved, the potential performance impacts, and the extreme infrequency of such modify DN operations, it is almost certain that any benefit that this could provide would be greatly outweighed by the associated costs. Nevertheless, support for modify DN operations for non-leaf entries will require a significant effort to ensure that referential integrity is properly maintained.

Synchronization and Change Notification Mechanisms

The synchronization mechanism should be simple to deploy even in the case of large deployments. When adding or removing a server, it should not be necessary to change the configuration of the other servers. Further, one should never have a constraint on the number of writable servers when designing a deployment. Therefore the synchronization mechanism must not put a limit on the number of writable server instances and it must not require that each writable instance have a reference to the other writable instances.

A good way to achieve this is to implement the synchronization around a centralized change notification system.

This system must have the following characteristics:

- It must be a central service so that each Directory Server in the environment will only need to be configured to communicate with it in order to ensure that changes are properly applied throughout the environment. It should not require that any of the server instances are aware of each other.

- It must be possible to configure this as a highly-available service so that there is no single point of failure. Multiple instances of the synchronization daemon should communicate with each other in a tightly-consistent manner so that any change accepted by one synchronization daemon will be atomically applied to all synchronization daemons.

- There must not be a limit imposed on the number of master (writable) instances of the server, nor should there be any requirement (or benefit) for all clients to send changes to a single instance. It should still be possible to configure a server to operate in read-only mode if that is desired for some reason, and in that case any write attempts should either be rejected or redirected to another instance that is writable.

- It must minimize the amount of state information that must be stored in the directory data itself. In particular, conflict detection and resolution algorithms may add a significant amount of state information as operational attributes within an entry. Given that conflicts should be relatively rare, especially in an environment using a centralized synchronization daemon, it should be better to take more expensive action to resolve a conflict when it is detected rather than maintaining a large amount of extra state information in each backend.

- It must provide the ability to notify other Directory Server instances of changes to the server configuration in addition to the data in actual backends. However it should be possible to apply this in a scoped manner because some configuration changes may be specific to a single server or group of servers.

- It must provide the ability to make certain kinds of changes atomic throughout the environment so that they will take effect immediately across all servers. This may include events like account lockout or disablement or changes to certain attributes declared "globally unique".

- It should provide the ability to define priority and quality of service for changes. The introduction of priority will allow certain types of changes (e.g., changes to certain attributes, or possibly certain types of LDAP operations) to receive preferential treatment so that they may be synchronized before other types, and quality of service could be used to limit the rate at which changes are processed or to only process higher priority changes for a period of time.

- It must provide the ability for servers to include subsets of the data, both in terms of the set of entries that they contain (i.e., filtered replication), and the individual attributes contained in those entries (i.e., fractional replication). Further, a server that does not contain the entire set of data must still be allowed to process changes to the data that it does contain (as well as add new entries or remove existing entries). This should make it possible to have scenarios in which no server in the environment contains a complete set of data (e.g., servers A and B have entries and/or attributes for applications X and Y, while servers C and D have entries and/or attributes for applications X and Z).

- It must provide an LDAP-accessible changelog service that is both globally ordered and highly available. The format of change records must be consistent with the changeLogEntry format in widespread use.

- It should provide the ability to utilize an assured replication mechanism, in which some set of servers have confirmed that they have seen and applied the change before a response is returned to the client. The set of servers in this case can be either a minimum number of systems (e.g., applied on at least two other systems) or a specified set of systems (e.g., applied on servers A, B, and C).

- It must be possible for entities other than Directory Server instances to register with this environment to receive notification of changes. It may also be possible under certain circumstances to allow them to publish their own change information so that it will be applied to all directory server instances.

- It must be possible to define access controls for the change notification environment in order to control what each registered system may do. These controls may be used to dictate whether a system may have read and/or write access, as well as for enforcing restrictions on the set of information to which that access applies.

- It must be possible to synchronize the schema definitions as necessary between all instances of the server. The original file structure for schema files should be preserved, and it should be possible to have more than one schema file that may be updated with the server online.

Ideally, the Directory Server environment will use a publish/subscribe model for this information. Whenever a change is made in the local database, each server instance can decide whether it is a change that should be published (and therefore visible to other servers), and each server can also decide whether to apply part, all, or none of each change performed elsewhere. In general, this can make filtered and fractional replication relatively simple, but there can still be complexities if a client sends an add or modify request to a server containing only a subset of the data, and that request includes information not appropriate for that particular instance. This can be dealt with in various ways, including rejecting the request, accepting the request and storing that additional information in the local instance, or redirecting it to a server containing the entire set of data. Given that this should be a relatively rare event and that it is desirable to minimize the knowledge of other servers in the environment, rejecting the change request might be the best approach.

Note that merely providing a publish/subscribe model for data synchronization may not be sufficient in all cases. For example, if a remote system containing only a small subset of the data is only accessible over a low-bandwidth connection, then it may not be feasible for it to see all changes in the directory environment. In this case, a mechanism that only transfers information about those changes that actually apply to the information in that remote system may be preferable. In this case, an intermediate subscriber on a higher-bandwidth network can consume and filter out all messages that don't apply and only forward on those that do.

Another significant concern that needs to be addressed is that of interoperability with existing replication models. It will be necessary to have at least minimal support for participating in such an environment in the course of migrating to OpenDS. Most likely, the simplest approach to this will be to consume the changes from a changelog and submit changes as a standard LDAP client. However, this will have to be investigated in much more detail to ensure proper coverage while a migration is in progress.

Access Control Rules

Because of the frequent use of the Directory Server as a source of authentication and authorization, the security of the environment is critical. The OpenDS server must provide flexible access controls, that are fast to evaluate and use a simple syntax.

In some existing implementations, access control rules are defined in a multi-valued ACI attribute which can be placed in any entry and apply to that entry and all its children (or potentially only to a subset of its children). It can be difficult to limit the scope of the ACI, and the syntax can be difficult to remember and can be prone to mistakes due to a missing parenthesis or semicolon. Rather than using this approach, the OpenDS server will use a new format for access control definitions in which each rule is defined in its own entry. For example:

```
dn: cn=Example administrator access, cn=everything, dc=example, dc=com
objectClass: top
objectClass: subentry
objectClass: ds-authz-access-control
objectClass: ds-authz-permission
objectClass: ds-authz-read-entry-permission
objectClass: ds-authz-add-entry-permission
objectClass: ds-authz-delete-entry-permission
objectClass: ds-authz-modify-entry-permission
objectClass: ds-authz-read-attributes-permission
objectClass: ds-authz-compare-attributes-permission
objectClass: ds-authz-write-attributes-permission
cn: Example administrator access
subtreeSpecification: { }
ds-authz-user: uid=admin,dc=example,dc=com
ds-authz-group: cn=Directory Administrators, dc=example, dc=com
ds-authz-all-user-attributes: true
ds-authz-all-operational-attributes: true
```

The OpenDS server access control rules should provide all of the capabilities commonly supported by Directory Server access control mechanisms, as well as a number of additional features, including:

- Connection-based access controls that can be used to immediately terminate connections from unauthorized client systems. These will not be traditional access control rules in that they will be defined at the connection handler level and therefore can vary based on the protocol used to communicate with the server.

- Bind-based access controls that can be used to prevent certain users from binding under various conditions (e.g., client address, time of day, protocol, authentication method, etc.).

- Better support for restricting access to controls and extended operations, including possible restrictions based on the type of control (e.g., the maximum number of candidate entries that will be considered if the results are to be sorted). The mechanism to accomplish it should be obvious to implement, available for all controls and extended operations, and it should be possible to configure custom limits based on the type of control.

- The ability to restrict certain operations based on the protocol used and/or whether the underlying connection is secure. Whether a connection is secure will be determined by the connection handler and may vary over time based on the operations that the client performs (e.g., a client connection may initially be insecure if it connects over LDAP, but may transition to secure if it uses the StartTLS extended operation).

Because this represents a significant change in the format used by access controls in other Directory Server implementations, it must be possible to convert existing ACI definitions to the new form when migrating to OpenDS. If OpenDS supports all access control capabilities of an existing implementation, then it should be possible to translate the definitions with no loss in functionality.

Other Security Features

Access controls are critical to the security of the Directory Server, but there are many other security-related areas as well. One very important facility is that of logging and auditing, but that will be addressed in-depth in a subsequent section. However, other security related capabilities that should be included in the Directory Server include:

- Typically a Directory Server has a single "superuser" account called the root DN (for example "cn=Directory Manager"). This is the most powerful account in the server and is able to bypass restrictions that may be placed on other users. Due to the damage that this account could cause if it were to be accessible by unauthorized users, the form of authentication allowed for the root DN should not be limited to a password. It should be possible to use (and potentially require) stronger forms of authentication like SASL EXTERNAL (using a client certificate) or GSSAPI (using Kerberos).

- The Directory Server should allow multiple root DN accounts. Allowing only one such account is dangerous because it is likely that the task of administering the Directory Server will be shared among several people, and only having a single account for performing some of these functions can limit the ability of an audit to determine which particular administrator made a given change. Similarly, the integrity of that single account can be at risk if one of those administrators leaves the company or assumes a different role, requiring that the password be changed. Both of these problems could be solved by allowing multiple accounts each with different credentials that can be treated as the root DN.

- In some cases, certain administrators may only need access to the root DN to perform one or two particular functions (e.g., backing up the server). In this case, it would be beneficial if the individual privileges associated with the root DN could be enumerated and assigned to users so that they only have the level of access required to perform their appropriate tasks. This capability is much like least privilege support in Solaris 10, but can actually be even more powerful because its implementation in the Directory Server could be such that it is scoped and only applies to a particular subset of the server. For example, one privilege that could be assigned is that of being able to bypass access control restrictions that may be defined. This could be very useful for an administrator of a particular organization, but only for entries within that organization and nowhere else in the directory.

- The Directory Server should support a number of mechanisms for protecting user passwords, beyond those primarily based on the UNIX crypt and SHA-1 algorithms. The UNIX crypt algorithm has been found to too readily vulnerable to dictionary attacks and may also be susceptible to brute force attacks in a reasonable amount of time. The SHA-1 algorithm is more secure, but recent findings by cryptography researchers have shown that it has flaws as well. It should be possible to use stronger digest algorithms to secure these passwords, for example the SHA-2 family of algorithms or RIPE-MD.

- While cryptographic hashes provide very good mechanisms for storing passwords in a manner in which it is impossible to discover the original plaintext used to generate the hash, there may be cases in which it is desirable to store the user's password in a reversible format. For example, one such case is to enable the use of the SASL DIGEST-MD5 mechanism. This mechanism allows a user to perform password-based authentication in a form that does not require the password to be transferred to the server in the clear, but in order for it to be successful it requires that the server already know the clear-text value. The Directory Server should not require such passwords to be stored in clear-text with no obfuscation or encryption. Instead it should be possible to use reversible encryption algorithms like DES-EDE, AES, or Blowfish to securely store these passwords.

- Typically, a Directory Server stores the password in a single form (by default, salted SHA-1). However, there may be cases in which it could be useful to store the same password in multiple forms. For example, this could facilitate more easily and securely synchronizing password changes in other systems (e.g., for use with another server that stores passwords as MD5 digests). Similarly, certain poorly-designed applications may authenticate users by retrieving the hashed value and manually calculating it against a digest calculated from the password provided by the end user. In that case, the application may expect the password in a particular format in order to be able to perform the authentication.

- In some cases, although the Directory Server can serve as a source of identity for a user, it may be desirable to store the password in an external system (e.g., another directory or a POP or IMAP server). In that case, it should be possible to configure the server to perform a type of pass-through authentication such that an attempt to bind as that user will cause that password to be validated against a remote repository. This may optionally include a mechanism for storing that password locally on a successful authentication as a means of aiding the transition from that remote repository to the new Directory Server instance.

- The LDAPv3 specification defines an anonymous bind as one that uses simple authentication with no password. It also indicates that the bind DN is typically empty as well in this case, but that it is also possible for it to be non-empty (e.g., the DN of a valid user in the directory). This can actually be a security risk for poorly-written applications that simply try to authenticate a user by binding with the provided DN and password. If they do not check to see if the password is empty, then those applications could be fooled into believing that the connection had been authenticated as the user DN contained in the bind request rather than as an anonymous connection, which could potentially allow that application to grant the user more privileges than should be allowed. Security holes based on this behavior have been observed in real-world applications, and the Directory Server should offer some form of protection against it. In particular, it should be possible to configure the Directory Server to reject any bind attempt that contains a bind DN but an empty password. This should not cause any problems with most LDAP-enabled applications, but can prevent security risks in poorly-designed directory clients.

- The Directory Server should offer a set of checks that may be used to determine whether a password is acceptable for use. In particular, it should be able to enforce a minimum acceptable length, check a new password against a history of previous passwords, and ensure that the

password is not the same as a subset of the attribute values in the entry. It should be possible to perform additional checks, including ensuring mixed case or a minimum number of non-alphabetic and/or non-alphanumeric characters, and possibly checking against a known dictionary or difference from the current password. It should also be possible for end users to develop their own kinds of password acceptability tests for use in the server.

- The Directory Server should provide a capability to force users to change their passwords after they have been reset, and this should not be restricted to passwords changed by the root DN. Rather, it should be applicable for any case in which a user's password is changed by anyone other than that user.

- In addition to simple authentication using a password, the Directory Server must support certain SASL mechanisms. In particular, it must support EXTERNAL, DIGEST-MD5, and GSSAPI and CRAM-MD5. It will also contain an interface that may be used to define new SASL mechanisms for use with the server.

- In general, performing a write every time a user binds can severely degrade authentication performance, but there can be valid reasons for keeping track of things like the last time that a user authenticated. However, if this feature is desired, it may be possible to implement without requiring a write on each bind. For example, if it is only necessary to record this information to the nearest day rather than to the nearest second, then the user's account would only need to be once per day, and any other binds performed on that day could skip that update. Potentially setting this to a user-defined format string could allow administrators that wish to use this feature the ability to control how fine-grained the updates will be.

- It should be possible to enforce password expiration in the Directory Server to ensure that users will not be able to authenticate unless they periodically change their passwords. It should also be possible to lock out or deactivate accounts that have not authenticated at all for longer than a specified length of time, provided last login time tracking is available.

- Whenever a user authenticates to the Directory Server with a password that will expire in the near future, then the server can include a special control in the bind response to provide a warning. However, many clients are not aware of and/or do not check for this response control and therefore can not make the end user aware of the situation. This problem could be addressed in a more reliable manner if the Directory Server were to send an e-mail message to the user the first time the warning control is included in the request. Further, additional forms of notification may also be useful, including if the password does expire, if an account is locked or unlocked, if an account is activated or deactivated, or if a password is reset by an administrator.

- In an application that needs to authenticate users, the best way to verify the credentials for a user is to perform a bind as that user. However, this can be inconvenient for applications that use connection pooling (which is recommended for performance reasons) because it would require either maintaining a separate pool of connections for binds, or performing unnecessary rebinds while also ensuring that no other operations are in progress on that connection until the bind has completed. A better solution would be to provide a mechanism for bind operations to

be processed without actually changing the authentication associated with the client and without preventing other operations from being processed concurrently on the same connection.

- Typically, whenever a user's password has expired, that user is prevented from authenticating and cannot do anything at all until an administrator has reset the password. Many administrators have indicated that it would be useful to have the ability for an end user to authenticate with an expired password but only be allowed to change that password and not perform any other action until that has been done. This does potentially carry risks for clients that only use the Directory Server as a repository for authentication (i.e., performing binds) and do not understand the controls returned by the server indicating that the password is expired. However, that could potentially be addressed by requiring a special control to be included with the bind request, or by using the password modify extended operation to change the password anonymously while providing both the existing and new passwords in the request.

- The proxied authorization control is extremely helpful for applications that perform connection pooling to use connections authenticated as one user to perform operations using the access granted to another. The implementation must properly check to determine whether the specified user has an expired password, an account that is locked out, or if there is any other reason that would prevent a user from binding directly. Further, certain resource limits associated with that user (e.g., size limit, time limit, and lookthrough limit) must be honored when operations are performed as that user via the proxied authorization control. This will eliminate problems in which a user might be allowed to perform some operations via the proxied authorization control that would not have been allowed if the user had attempted to authenticate directly, or result in different limits being enforced for that user that could unnecessarily restrict capabilities or allow inefficient requests to consume more resources than should have been allowed.

- The Directory Server should offer the ability to perform attribute encryption, which can help protect the database when stored on-disk. However, there may be cases in which it is desirable to encrypt the entire database. Further, an attribute encryption mechanism cannot always be used to protect that data as some types of operations require that the information be exported to LDIF in an de-encrypted form. It may be possible to provide complete support for encryption within the database in a form that would never require that information to be exposed on disk in clear-text.

- Another potential problem with encrypted attribute support is that it only extends to the on-disk storage of the data. It does nothing to prevent the de-encrypted data from being exposed over the network. For certain kinds of operations, it may be that the clear-text value may never actually be needed except for confirmation that a user-provided value matches what is in the directory. In those cases, it could be very useful to configure certain attributes so that the value always remains hashed or encrypted but it may still be possible for the server to allow compare operations or base-level equality searches to work by comparing the clear-text value provided by the client with the hashed or encrypted value stored in the directory.

Logging and Alerting Mechanisms

The logging subsystem in the Directory Server is another crucial component because it has such a large number of uses. It provides information that can be used for debugging problems, creates audit trails that may be required for regulatory compliance, and offers means of obtaining performance metrics and usage statistics. However, there are a number of features required in order for logging to be used to its fullest potential. Some of these features include:

- It must be possible to aggregate the log information together in a common location so that it may be more easily processed or examined. This may come in many different forms, but in many cases the intent is to either send the log messages to a centralized logging service in real-time (or near real-time in some cases), or to have them written to a network-accessible database. Storing the information in a database can have a number of benefits, particularly if it is broken up into individual components that can be easily queried to create reports, but it can also add a significant amount of complexity to the environment. A centralized service that accepts the messages and writes them to a common log file is much simpler, but it could be more difficult for administrators to be able to extract the desired information from it.

- It should be possible to use cryptographic hashing and/or digital signatures to generate log information that is tamper evident. Otherwise the log files are still very useful for debugging information, but auditing cannot be guaranteed because it would be possible for someone to tamper with their contents (e.g., to add, remove, or change records) without detection. If digital signatures are used in a way such that each line contains a signed hash of the contents of both the current line and immediately preceding line then if any change had been made to the contents of the file the associated signatures would be unverifiable. Alternately, rather than signing each line of the log it could be possible to sign small chunks of the file so that there would be a less notable performance impact but still a relatively small set of messages that would be in question if the signature could not be verified. It may be desirable to use HMAC (hashed message authentication code) operations rather than digital signatures for improved performance, but the complexity of such an implementation (and in particular, additional key management overhead) should be considered in making this decision. In any event, a tool must be provided with the Directory Server that can be used to verify the authenticity and integrity of the log information.

- It should be possible to filter out (i.e., not log) messages based on certain criteria. For example, some users complain about the unnecessary volume of connection establishments and closures when they have a hardware load balancer that establishes a TCP connection to the server every few seconds to verify that it is still running. In such cases it could be desirable to suppress log messages related to connection and disconnection from that load balancer. Similarly, if a user is only interested in monitoring changes to the server then it could be useful to suppress search, compare, and bind operations. Note that for cases in which both filtering and signing are enabled, the process of filtering is performed before the signing would occur, both to improve performance and to prevent invalid signatures due to missing information in the log.

- The Directory Server should have three primary types of loggers: access, error, and debug. Access loggers will be used to record information about events that occur within the server related to client processing. Error loggers will be used to record information about errors, warnings, notices, and significant events that occur within the server. Debug loggers will be used to record messages used for debugging problems and will only be active if the server is operating in a debug mode.

- It should be possible to activate multiple concurrent loggers of each type within the server. Each logger may perform different kinds of filtering, use different output formats, and may write to different repositories. This can be used, for example, to have separate logs for operations that may be of particular interest to administrators (e.g., one log containing all operations, one containing only changes to the server, one containing only operations that did not complete successfully, etc.). A documented API should be made available to allow users to develop their own loggers for use with the server.

- Each message that may be written to an error log must be associated with a unique integer value. All possible error messages must be documented. For messages used to indicate a warning or error, that documentation should also include information about any action that should be taken if that condition arises.

- Each message that may be written to an error log must be encoded in a form that could allow for the localization of that message. This should be in the form of a format string which can be dynamically populated with information specific to the event that has occurred so that the ability to perform localization does not interfere with the clarity or usefulness of the message.

- Each message that may be written to an error log must be associated with both a category (i.e., the component of the Directory Server in which the message is logged) and a severity. Error loggers may be configured to filter messages using these characteristics.

- The Directory Server should be provided with one or more error loggers that may serve as alerting mechanisms. For example, it may be desirable for any severe errors encountered within the server to send an e-mail message or page to an administrator, generate an SNMP trap, and/or actively notify a third-party monitoring system. It may also be desirable to implement features in applications that can use the Directory Server so that they can receive such notifications and act appropriately (e.g., if a Directory Proxy Server is alerted that a Directory Server instance is shutting down, then it will be able to adjust its routing policies to avoid sending requests to that server until it receives notification that it is again available for use).

- All methods used to perform debug logging must return a `boolean` value of "`true`". All calls to these methods must be immediately preceded by the `assert` keyword. In this way, debug messages will only be written if assertions have been enabled in the JVM and will have no performance impact if they are not needed. This makes it possible to have a single release of the server that may be used for production operation but that may also be used for in-depth debugging if the need arises. An additional benefit is that the JVM may allow enabling assertions in only a subset of classes, and therefore further restrict any overhead that debugging may introduce.

- Debug log messages do not need to allow for localization, nor do they need to be assigned unique numbers. However, they should be associated with both a category and a severity so that debug loggers may filter messages based on category and/or severity.

- The Directory Server should be provided with tools that may be used to analyze access log information in order to allow administrators to better understand the use of the server in their environment. Such tools may provide functions like reporting usage and performance metrics, identifying potential errors, providing tuning recommendations, etc.

The Plugin API

The plugin API is the primary mechanism for end users and third parties to extend and customize the operation of the Directory Server. Because of this, the API must be very stable to ensure that plugins written for one version of the Directory Server will continue to work in future versions without changes or re-compilation (although it may be desirable to update the code in order to take advantage of new features). There will also be a number of project-developed plugins to provide significant portions of the server functionality.

The definition of what exactly constitutes a "plugin" may be somewhat blurred because of the extensible nature of the Directory Server and the fact that most of its components are written to implement some abstract API. However, the primary distinguishing characteristics for a plugin are that it may be used to implement some form of fringe processing that is not essential to the operation of the Directory Server, and that there are distinct, well-defined times in the life cycle of a client request, client connection, and/or the Directory Server process at which they may be invoked. Nevertheless, it may be common to use the term "plugin" to any extensible component of the Directory Server, particularly because the term is applied in that manner to other Directory Server implementations.

Typically a Directory Server provides two primary types of plugins for most kinds of operations: pre-operation and post-operation. Pre-operation plugins are called before primary processing is done for an operation, and post-operation plugins are called as one of the last steps in processing an operation. However, there are problems with having only these two types of plugins, primarily because they are both called too late in the process. For example, post-operation plugins are called after a response has already been sent to the client, whereas there are times that it would be nice to be able to perform additional processing before the result is sent to the client. Similarly, pre-operation plugins are called after a certain amount of critical processing is performed (e.g., the selection of the backend(s) to use to handle a request), whereas it may be desirable for a plugin to change elements of the request that could have already been used in that earlier processing. Therefore, the OpenDS server should provide four types of operation plugins:

- Pre-parse plugins should be called immediately after a worker thread has taken the request from the work queue. It must not be able to modify the message ID or protocol operation type, but it may alter any aspect of the protocol operation (with the exception of the message ID of the target operation for an abandon request, since message ID's may not be altered), and it may add, remove, or modify any controls associated with the request. Pre-parse plugins should always be called for all requests, but a pre-parse plugin may indicate that no further processing should be performed on the request.

- Pre-operation plugins should be called after all preliminary processing has been completed and immediately before the core processing is started. Pre-operation plugins may not be called if any problems are found in the preliminary processing (e.g., the targeted entry does not exist or access controls would not allow the requested operation).

- Post-operation plugins should be called immediately after the core processing is completed for a request and before any response is sent to the client. Post-operation plugins will be called regardless of whether the operation was successful, although they may prevent any further processing on the request.

- Post-response plugins should be called as the final set of processing for a request and may be used for any processing that should be performed after an operation that does not need to be completed before the response is sent. This can help decrease the response time to the client, but will still tie up the worker thread until processing is complete. Post-response plugins will be called regardless of whether a successful response was sent to the client, unless the post-operation plugins indicate that no further processing should be performed.

Note that not all of these types of plugins may apply for all types of operations. For example, neither abandon nor unbind operations have an associated response, so there is no need for a post-response plugin. The unbind request protocol operation does not contain any elements and therefore it may be natural to think that they do not require a pre-parse plugin type, but it is theoretically possible to include controls in the request and therefore pre-parse plugins may operate upon them. The same is true for abandon requests whose only element is the message ID of the operation to abandon, which may not be altered.

There may also be other types of plugins used in the Directory Server. Some of these may include:

- Post-connect plugins, which are called after a connection has been established to the Directory Server. They may terminate the connection if it is deemed appropriate.

- Post-disconnect plugins, which are called after a connection to the Directory Server has been closed (regardless of whether that closure was initiated by the client or the server).

- Search result entry plugins, which will be called before returning a search result entry to a client. They may alter the contents of that entry or any associated controls, may prevent that entry from being returned to the client, or may end processing on the request.

- Search result reference plugins, which will be called before returning a search result reference (i.e., a referral) to a client. They may alter the contents of that reference or any associated controls, may prevent it from being returned to the client, or may end processing on the request.

- Intermediate response plugins, which will be called before returning an intermediate response message to a client. They may alter the contents of that response or any associated controls, may prevent it from being returned to the client, or may end processing on the request.

- Server startup plugins, which are invoked during the course of the Directory Server startup process. They will be loaded and invoked prior to the registration of any other type of plugin.

- Server shutdown plugins, which are invoked during the course of a graceful termination of the Directory Server process (and will likely not be invoked in the event of an abnormal termination

such as a crash within the JVM, a forceful termination of the Java process, or a lower-level problem with the hardware or operating system). They will be called after all connection handlers have been stopped, client connections terminated, and operations in progress have been abandoned, but before any other processing associated with the shutdown has been performed. Server shutdown plugins must complete quickly and must not themselves call `System.exit()`, `Runtime.exit()`, or `Runtime.halt()`, or perform any other operation that may short-circuit the shutdown process.

- LDIF import plugins, which are invoked during the course of reading entries from an LDIF file. They may alter the contents of the entries or prevent entries from being imported.

- LDIF export plugins, which are invoked during the course of writing entries to an LDIF file. They may alter the contents of the entries or prevent entries from being exported.

The plugin API provided by the Directory Server must be inherently safe. That is, it must not be possible for a faulty plugin to crash the Directory Server. For those plugins that are written entirely in Java, this should be relatively simple to accomplish by ensuring that appropriate `try`/`catch` blocks are defined. It may not be possible to ensure this level of safety if any native code is used by the plugin, and therefore it is strongly recommended that no native code is used in plugins unless it is absolutely essential. Plugins must not call `System.exit()`, `Runtime.exit()`, or `Runtime.halt()`, nor may they perform any other operation that can prematurely terminate the Java process. A method may be made available to plugins to initiate a graceful shutdown of the Directory Server if that plugin determines that it is appropriate to do so, but this should be used only under extremely limited circumstances, and any time it is used every attempt should be made to ensure that the reason for the shutdown is clearly logged or otherwise made available to administrators.

Supported Controls and Extended Operations

One of the primary improvements in LDAPv3 over the previous version is the addition of features that make it possible to enhance the protocol in manners not necessarily anticipated at the time the specification was developed. This extensibility comes in two primary forms: controls and extended operations. A control is an element that may be included in any request or response to serve as an indication of additional processing that should be performed, while an extended operation is a separate type of protocol operation that can be used to perform some type of processing completely unrelated to any of the standard operation types. Since then, a number of types of controls and extended operations have been defined covering a variety of areas.

The set of controls that the Directory Server may process is somewhat limited because there is no straightforward way to implement support for new controls in a way that will work in general cases. Some types of controls may be implemented using plugins, but in general the set of controls that may be used will be hard-coded into the server. As such, it is important to ensure the set of provided controls is relatively comprehensive. Some of the controls that should be supported include:

- The manage DSA IT control, which is used to indicate that referrals should be returned as LDAP entries rather than as referrals.
- The server-side sort control, which is used to request that the server sort a set of search results before returning them to the client.
- The paged results control, which is used to allow clients to retrieve a set of search results in "pages" consisting of specified number of entries. Only sequential iteration through the results is allowed.
- The virtual list view (VLV) control, which is used to allow clients to request specific subsets of entries matching search criteria. This is similar to the paged results control, but allows arbitrary sets of entries to be returned.
- The persistent search control, which can be used to identify changes in the directory involving entries that match a specified set of criteria.
- The proxied authorization control, which can be used to request that an operation be processed using the rights associated with one user while authenticated as another.
- The get effective rights control, which can be used to retrieve information about the permissions that a user has when dealing with a specified entry.

- The authorization identity request control, which can be used to retrieve information about the currently authenticated user.
- The Netscape password expired and password expiring controls, which can indicate whether a user's password is currently expired or will expire in the near future.
- The proposed-standard password policy control, which can provide information about certain aspects of an account status, including whether the password is expired or expiring, whether the password needs to be reset, whether an account has been locked out, or whether a provided password does not meet configured requirements.
- The LDAP no-op control, which can be used to indicate that all processing associated with the operation should be performed but no changes should be made in the directory.
- The matched values control, which can be used to indicate that only the attribute values matching a provided set of criteria should be returned.
- The LDAP assertions control, which can be used to perform an operation only if the target entry matches a given assertion.
- The LDAP read entry controls, which can be used to retrieve the contents of an entry either immediately before or immediately after an update.

The set of extended operations that may be supported is much more flexible because they can easily be implemented in an extensible manner. Therefore, additional extended operations may be added by users or other third parties. However, the support for the following controls should be included with the server:

- The notice of disconnection unsolicited notification, which can be used to indicate that the server will be closing the connection to the client and may include a reason.
- The "Who am I?" extended operation, which can be used to provide information about the currently-authenticated client.
- The password modify extended operation, which can be used to change a password for a user, optionally including the user's current password for verification.
- The cancel extended operation, which is similar to the abandon protocol operation but includes a response to indicate whether or not the operation was canceled and may require that a response be returned for the request that was canceled.
- The StartTLS extended operation, which may be used to initiate a secure communication channel over an otherwise insecure connection.

Common Development and Distribution License, Version 1.0

Unless otherwise noted, all components of the OpenDS Directory Server, including all source, configuration, and documentation, are released under the Common Development and Distribution License (CDDL), Version 1.0. The full text of that license is as follows:

1. Definitions.

1.1. "Contributor" means each individual or entity that creates or contributes to the creation of Modifications.

1.2. "Contributor Version" means the combination of the Original Software, prior Modifications used by a Contributor (if any), and the Modifications made by that particular Contributor.

1.3. "Covered Software" means (a) the Original Software, or (b) Modifications, or (c) the combination of files containing Original Software with files containing Modifications, in each case including portions thereof.

1.4. "Executable" means the Covered Software in any form other than Source Code.

1.5. "Initial Developer" means the individual or entity that first makes Original Software available under this License.

1.6. "Larger Work" means a work which combines Covered Software or portions thereof with code not governed by the terms of this License.

1.7. "License" means this document.

1.8. "Licensable" means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

1.9. "Modifications" means the Source Code and Executable form of any of the following:

A. Any file that results from an addition to, deletion from or modification of the contents of a file containing Original Software or previous Modifications;

B. Any new file that contains any part of the Original Software or previous Modifications; or

C. Any new file that is contributed or otherwise made available under the terms of this License.

1.10. "Original Software" means the Source Code and Executable form of computer software code that is originally released under this License.

1.11. "Patent Claims" means any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent Licensable by grantor.

1.12. "Source Code" means (a) the common form of computer software code in which modifications are made and (b) associated documentation included in or with such code.

1.13. "You" (or "Your") means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

2. License Grants.

2.1. The Initial Developer Grant.

Conditioned upon Your compliance with Section 3.1 below and subject to third party intellectual property claims, the Initial Developer hereby grants You a world-wide, royalty-free, non-exclusive license:

(a) under intellectual property rights (other than patent or trademark) Licensable by Initial Developer, to use, reproduce, modify, display, perform, sublicense and distribute the Original Software (or portions thereof), with or without Modifications, and/or as part of a Larger Work; and

(b) under Patent Claims infringed by the making, using or selling of Original Software, to make, have made, use, practice, sell, and offer for sale, and/or otherwise dispose of the Original Software (or portions thereof).

(c) The licenses granted in Sections 2.1(a) and (b) are effective on the date Initial Developer first distributes or otherwise makes the Original Software available to a third party under the terms of this License.

(d) Notwithstanding Section 2.1(b) above, no patent license is granted: (1) for code that You delete from the Original Software, or (2) for infringements caused by: (i) the modification of the Original Software, or (ii) the combination of the Original Software with other software or devices.

2.2. Contributor Grant.

Conditioned upon Your compliance with Section 3.1 below and subject to third party intellectual property claims, each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license:

(a) under intellectual property rights (other than patent or trademark) Licensable by Contributor to use, reproduce, modify, display, perform, sublicense and distribute the Modifications created by such Contributor (or portions thereof), either on an unmodified basis, with other Modifications, as Covered Software and/or as part of a Larger Work; and

(b) under Patent Claims infringed by the making, using, or selling of Modifications made by that Contributor either alone and/or in combination with its Contributor Version (or portions of such combination), to make, use, sell, offer for sale, have made, and/or otherwise dispose of:

(1) Modifications made by that Contributor (or portions thereof); and (2) the combination of Modifications made by that Contributor with its Contributor Version (or portions of such combination).

(c) The licenses granted in Sections 2.2(a) and 2.2(b) are effective on the date Contributor first distributes or otherwise makes the Modifications available to a third party.

(d) Notwithstanding Section 2.2(b) above, no patent license is granted: (1) for any code that Contributor has deleted from the Contributor Version; (2) for infringements caused by: (i) third party modifications of Contributor Version, or (ii) the combination of Modifications made by that Contributor with other software (except as part of the Contributor Version) or other devices; or (3) under Patent Claims infringed by Covered Software in the absence of Modifications made by that Contributor.

3. Distribution Obligations.

3.1. Availability of Source Code.

Any Covered Software that You distribute or otherwise make available in Executable form must also be made available in Source Code form and that Source Code form must be distributed only under the terms of this License. You must include a copy of this License with every copy of the Source Code form of the Covered Software You distribute or otherwise make available. You must inform recipients of any such Covered Software in Executable form as to how they can obtain such Covered Software in Source Code form in a reasonable manner on or through a medium customarily used for software exchange.

3.2. Modifications.

The Modifications that You create or to which You contribute are governed by the terms of this License. You represent that You believe Your Modifications are Your original creation(s) and/or You have sufficient rights to grant the rights conveyed by this License.

3.3. Required Notices.

You must include a notice in each of Your Modifications that identifies You as the Contributor of the Modification. You may not remove or alter any copyright, patent or trademark notices contained within the Covered Software, or any notices of licensing or any descriptive text giving attribution to any Contributor or the Initial Developer.

3.4. Application of Additional Terms.

You may not offer or impose any terms on any Covered Software in Source Code form that alters or restricts the applicable version of this License or the recipients' rights hereunder. You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Software. However, you may do so only on Your own behalf, and not on behalf of the Initial Developer or any Contributor. You must make it absolutely clear that any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.5. Distribution of Executable Versions.

You may distribute the Executable form of the Covered Software under the terms of this License or under the terms of a license of Your choice, which may contain terms different from this License, provided that You are in compliance with the terms of this License and that the license for the Executable form does not attempt to limit or alter the recipient's rights in the Source Code form from the rights set forth in this License. If You distribute the Covered Software in Executable form under a different license, You must make it absolutely clear that any terms which differ from this License are offered by You alone, not by the Initial Developer or Contributor. You hereby agree to indemnify the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of any such terms You offer.

3.6. Larger Works.

You may create a Larger Work by combining Covered Software with other code not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License are fulfilled for the Covered Software.

4. Versions of the License.

4.1. New Versions.

Sun Microsystems, Inc. is the initial license steward and may publish revised and/or new versions of this License from time to time. Each version will be given a distinguishing version number. Except as provided in Section 4.3, no one other than the license steward has the right to modify this License.

4.2. Effect of New Versions.

You may always continue to use, distribute or otherwise make the Covered Software available under the terms of the version of the License under which You originally received the Covered Software. If the Initial Developer includes a notice in the Original Software prohibiting it from being distributed or otherwise made available under any subsequent version of the License, You must distribute and make the Covered Software available under the terms of the version of the License under which You originally received the Covered Software. Otherwise, You may also choose to use, distribute or otherwise make the Covered Software available under the terms of any subsequent version of the License published by the license steward.

4.3. Modified Versions.

When You are an Initial Developer and You want to create a new license for Your Original Software, You may create and use a modified version of this License if You: (a) rename the license and remove any references to the name of the license steward (except to note that the license differs from this License); and (b) otherwise make it clear that the license contains terms which differ from this License.

5. DISCLAIMER OF WARRANTY.

COVERED SOFTWARE IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED SOFTWARE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE COVERED SOFTWARE IS WITH YOU. SHOULD ANY

COVERED SOFTWARE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

6. TERMINATION.

6.1. This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

6.2. If You assert a patent infringement claim (excluding declaratory judgment actions) against Initial Developer or a Contributor (the Initial Developer or Contributor against whom You assert such claim is referred to as "Participant") alleging that the Participant Software (meaning the Contributor Version where the Participant is a Contributor or the Original Software where the Participant is the Initial Developer) directly or indirectly infringes any patent, then any and all rights granted directly or indirectly to You by such Participant, the Initial Developer (if the Initial Developer is not the Participant) and all Contributors under Sections 2.1 and/or 2.2 of this License shall, upon 60 days notice from Participant terminate prospectively and automatically at the expiration of such 60 day notice period, unless if within such 60 day period You withdraw Your claim with respect to the Participant Software against such Participant either unilaterally or pursuant to a written agreement with Participant.

6.3. In the event of termination under Sections 6.1 or 6.2 above, all end user licenses that have been validly granted by You or any distributor hereunder prior to termination (excluding licenses granted to You by any distributor) shall survive termination.

7. LIMITATION OF LIABILITY.

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL YOU, THE INITIAL DEVELOPER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED SOFTWARE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS EXCLUSION AND LIMITATION MAY NOT APPLY TO YOU.

8. U.S. GOVERNMENT END USERS.

The Covered Software is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" (as that term is defined at 48 C.F.R. 252.227-7014(a)(1)) and "commercial computer software documentation" as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Software with only those rights set forth herein. This U.S. Government Rights clause is in lieu of, and supersedes, any other FAR, DFAR, or other clause or

provision that addresses Government rights in computer software under this License.

9. MISCELLANEOUS.

This License represents the complete agreement concerning subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This License shall be governed by the law of the jurisdiction specified in a notice contained within the Original Software (except to the extent applicable law, if any, provides otherwise), excluding such jurisdiction's conflict-of-law provisions. Any litigation relating to this License shall be subject to the jurisdiction of the courts located in the jurisdiction and venue specified in a notice contained within the Original Software, with the losing party responsible for costs, including, without limitation, court costs and reasonable attorneys' fees and expenses. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License. You agree that You alone are responsible for compliance with the United States export administration regulations (and the export control laws and regulation of any other countries) when You use, distribute or otherwise make available any Covered Software.

10. RESPONSIBILITY FOR CLAIMS.

As between Initial Developer and the Contributors, each party is responsible for claims and damages arising, directly or indirectly, out of its utilization of rights under this License and You agree to work with Initial Developer and Contributors to distribute such responsibility on an equitable basis. Nothing herein is intended or shall be deemed to constitute any admission of liability.

NOTICE PURSUANT TO SECTION 9 OF THE COMMON DEVELOPMENT AND DISTRIBUTION LICENSE (CDDL)

For Covered Software in this distribution, this License shall be governed by the laws of the State of California (excluding conflict-of-law provisions).

Any litigation relating to this License shall be subject to the jurisdiction of the Federal Courts of the Northern District of California and the state courts of the State of California, with venue lying in Santa Clara County, California.