

# Securing Distributed Erlang

Dave Smith (dizzyd@gmail.com)

December 1, 2008

## Overview

The protocol that enables Distributed Erlang exhibits a number of security flaws that prevent it from being used across unsecured networks. Specifically, it exhibits the following deficiencies:

1. *All-or-nothing security*: Once a connection is created between two nodes there is no way to restrict access to individual portions of the runtime or running code.
2. *No transport-level security*: Messages are exchanged between nodes over a plain-text TCP connection. As such, they are vulnerable to a variety of attacks, including forgery, interception and tampering.
3. *Limited authentication*: Authentication of individual connections can only be done via a single shared secret system. All nodes must use the same shared secret, and thus the entire network of nodes is vulnerable if the any one of them is compromised.

This project describes an alternative protocol, `SDIST` (Secure Distributed Erlang) that provides the necessary infrastructure to secure and protect exchanges between Erlang nodes in untrusted environments. `SDIST` integrates support for SASL, an Internet standard meta-protocol for authentication and security. The protocol implementation also provides a filtering layer to restrict what messages may be sent between two nodes, thus restricting access to individual portions of the runtime and/or running code.

## 1 Erlang Overview

### 1.1 Concepts

Erlang is a concurrent language and runtime designed for building fault-tolerant, massively concurrent systems. Originally developed by Ericsson as a proprietary

language for building telephone switches, the Erlang language and runtime is now available as an Open Source codebase [2]. Recently it has enjoyed rapid adoption by many software companies, including Amazon and T-Mobile for use in Internet related projects which benefit from the robust reliability and concurrency mechanisms it provides.

Erlang derives its concurrent nature from a “process” centric design. An Erlang `PROCESS` is a lightweight thread of execution that exchanges messages with other processes. Message passing is the only mechanism available for cross-process communication – no state is shared among processes. This design enables any one process to fail without causing the system to enter an indeterminate state.

The Erlang VM also provides the primitives necessary for processes to monitor each other. This takes two forms: `MONITORS` and `LINKS`. Monitoring is a mechanism that creates a one-way relationship between processes. For example, process A may create a monitor on process B; when process B fails or otherwise exits, process A will receive a notification. Links are a variation on monitors, where a two-way relationship is created between one or more processes that ensures the linked process(es) will exit when the other process terminates or crashes.

Erlang uses a unique approach to error handling which encourages a process to “crash” when an error occurs. This paradigm, when coupled with monitoring/linking of processes and no shared state permits the process to be automatically restarted on failure and enable the system to recover to a consistent state. The core idea is that a fault-tolerant computer system must have at least two computers which observe and restart each other as necessary [1].

Lightweight, independent processes combined with message passing, linking and monitoring, represent the foundation for building applications which can self-monitor for failure and deal gracefully with partial failures. In the scope of a single Erlang node, there can be hundreds of thousands processes active at any given moment. Distributed Erlang takes these ideas and makes them applicable across a network.

An instance of the Erlang VM is a `NODE`. Distributed Erlang enables a network of nodes to exchange messages, spawn, monitor and link processes across nodes and otherwise interoperate as if all processes are local to any given node. This functionality is a natural extension of Erlang and is facilitated by the absence of shared-state amongst processes. When a process wishes to exchange messages with another process, there is no need for either process to be “local” – with the proper addressing, the two processes can be distributed across independent nodes.

## 1.2 Distributed Erlang Flaws

“All-or-nothing security” is a way of describing the required implicit trust of all nodes which interconnect using Distributed Erlang. Once a connection is constructed between two nodes, the trust relationship is bidirectional and exhaustive. Either node may send a message to the other node, without restriction or validation. The existing RPC functionality which is built on top of Distributed Erlang permits

either node to invoke any function on the remote node. The nodes may terminate (or halt) each other, inspect any/all in-memory information available, terminate running applications within the remote node, start new processes which steal run time from the remote node, etc.

Distributed Erlang is further flawed in that it provides very limited transport-level security between nodes. SSL is supported for cross-node communication, but if enabled it must be enabled for *all* nodes. Furthermore, the use of SSL dictates that all nodes must perform full peer-to-peer certificate based authentication; this complicates deployment dramatically since all nodes must share a common Certificate Authority (at a minimum). Ideally, Distributed Erlang should support the “anonymous” SSL client authentication commonly used in web browsers which requires only a single party to be authenticated to perform useful interactions. However, with the all-or-nothing security approach, full peer-to-peer authentication makes sense as each party wants to strongly authenticate the identity of the other party.

Finally, Distributed Erlang has a very limited form of single-level authentication. It is not possible to construct a security gradient which grants varying levels of security, depending on the authenticated identity of a connection. The authentication system is built on a single shared-secret; all nodes which wish to interconnect must know that secret. This has the undesirable effect wherein a compromise of any one of the nodes means that *all* the other nodes have been compromised.

As currently implemented, Distributed Erlang is not suitable for use across the Internet, and can only be safely used on a secure, trusted network, where all nodes are trusted implicitly [4, 7]. In modern deployment terms, this means that it is unsafe to deploy an application within a network DMZ, where the network is only considered marginally more secure than the external Internet. These deficiencies also make it difficult to implement a Service-Oriented Architecture (SOA) using Distributed Erlang, since there is no clean way to restrict what portions of the service are available to untrusted clients without constructing a one-off protocol specific to the service. The SDIST protocol will systematically address each of the flaws described above.

## 2 Protocol

### 2.1 Overview

SDIST is a TCP-based, asynchronous binary protocol. The protocol is built using current Internet Engineering Task Force (IETF) specified authentication protocols to ensure that the identity of nodes is securely validated using current best practices. SDIST uses the SASL meta-protocol [9] to negotiate the best available authentication mechanism, while also ensuring that new forms of authentication can be added to protocol implementations without requiring protocol changes. SDIST also provides mechanisms for using the Transport Layer Security protocol [3] so that privacy and data integrity between nodes can be ensured. Finally, SDIST is constructed so

that fine-grained, per-message access-control-lists (ACLs) can be applied to reduce exposure between individual nodes.

Each packet is prefixed by a four-byte, network-order octet count – the theoretical upper limit for a single SDIST packet is thus  $2^{32-1}$  octets. It is recommended, however, that implementations impose an upper limit on allowed packets sizes to avoid DoS attacks.

A single SDIST packet has the following layout:

Name	Type	Octets	Description
Packet length	Integer, network-order	4	Number of octets, exclusive, that compose the packet
Control length	Integer, network-order	2	Number of octets, within the packet that compose the control message
Control message	External Term Format	n	Erlang term that specifies an action/response message
Data message	External Term Format	m	Erlang term that contains data associated with the control message

SDIST makes heavy use of the External Term Format [5] which is also used by Distributed Erlang. This serialization format enables nodes to exchange arbitrarily complex data structures (tuples, lists, atoms, numbers, etc.) in a manner that ensures backwards and forwards compatibility.

SDIST connections between two nodes utilize a uni-directional trust model. The client (the node initiating the connection) authenticates itself to the server (the node accepting the connection), but not vice versa. Operations such as spawning processes and performing RPCs are only executed by the server on request from the client; the client does not permit the server to attempt the same operations in the opposite direction.

## 2.2 Authentication

A significant portion of the SDIST protocol is devoted to the authentication process. As noted in the overview, SDIST is built using the SASL meta-protocol for maximum extensibility of authentication mechanisms. SASL compliance dictates that the protocol provide two pieces of essential infrastructure:

1. Mechanism negotiation
2. Challenge/Response exchanges

SDIST uses a simple process for negotiating the mechanism. After creating the TCP connection to the server, the first packet the client generates is a request for a list

of mechanisms. The server returns a list of mechanisms in the preferred order of usage. The client then selects the first available mechanism that it is configured for and attempts to authenticate using that mechanism; failure to authenticate then causes the client to try the next mechanism until all mechanisms are exhausted or authentication succeeds.

Once a mechanism is selected, the client transmits an initial authentication packet to the server. The server then responds with a challenge which the client then uses to further process authentication data and generate a response. This process continues until the selected authentication mechanism on the server is satisfied by the evidence presented by the client, or the authentication fails. In the success case, the client is notified that authentication is complete and it may begin transmitting operational requests to the server. In the case of failure, the client is notified that authentication has failed and may then attempt to authenticate using another mechanism. The server *should* take steps to ensure that the authentication process is terminated after a fixed amount of time, so as to conserve resources and permit other clients the opportunity to use the server.

Transport Layer Security (TLS) is represented as a formal mechanism within the protocol. The server advertises TLS using the `STARTTLS` mechanism identifier; the client then exchanges a corresponding message which is a signal to the server that the socket should begin negotiation of the TLS protocol using the connected socket. If the negotiation completes successfully, the client must then re-attempt mechanism negotiation and authentication. The server *may* make additional mechanisms available (such as plain-text authentication) to the client once the TLS connection is established. Furthermore, if full peer-to-peer authentication was performed as a part of the TLS protocol negotiation, the server may advertise the `EXTERNAL` mechanism identifier which uses the exchanged SSL certificates to complete the authentication process, per Appendix A in [9].

## 2.3 Operations

`SDIST` provides three core operations, once the connection has been authenticated:

1. `SPAWN`: starts a new process on the server node
2. `RPC`: invoke a module/function on the server node
3. `SEND`: deliver a message to a process active on the server node

Each of these operations are permitted by the server, contingent upon authorization validation by the server node. For both `SPAWN` and `RPC`, the server will generate a result message which provides the client with information regarding the outcome of the operation. `SEND` is slightly different, in keeping with normal Erlang “send-and-pray” delivery scheme. As with local messaging, if the application using `SDIST` wishes to guarantee message delivery it must do so by defining a reliable protocol atop the unreliable delivery mechanisms provided by `SDIST`.

Each operation *must* be validated by the server, prior to execution. This fine-grained execution control provides a way to restrict access to only those modules and functions that the server operator wishes to expose to a specific client.

The SPAWN and RPC messages include a tag which can be used by the client to connect responses back to an originating process in the client VM. The server should start operations in the order in which they are received, but responses to any given operation may be returned out of order (due to the asynchronous nature of cross process messaging).

## 2.4 Control Messages

The protocol defines the following control messages:

- `start_tls :: atom()`  
Sent by the client indicating that it wishes to switch to TLS (Transport Layer Security). The client *should* request a list of mechanisms from the server prior to requesting this.
- `ping :: atom()`  
Sent by either client or server to validate that the connection is still alive.
- `pong :: atom()`  
Sent in response to “ping” message from other side of connection.
- `mechanisms :: atom()`  
Sent from client to server when requesting a list of supported authentication and security mechanisms.
- `{ mechanisms, [binary()] } :: tuple()`  
Sent from the server to the client in response to the previous message. Contains a list of supported mechanisms, per the SASL specification.
- `{ auth_init, mechanism::binary(), data::binary() } :: tuple()`  
Sent from the client to the server to begin the process of authentication. May include data to reduce number of round trips, per the SASL specification, section 3.3.
- `{ auth_challenge, data::binary() } :: tuple()`  
Sent from the server to the client. The contents of data are mechanism specific.
- `{ auth_response, data::binary() } :: tuple()`  
Sent from the client to the server as a response to challenges. The contents of data are mechanism specific.
- `auth_ok :: atom()`  
Sent from the server to the client once authentication has completed successfully.

- `auth_failed :: atom()`  
Sent from the server to the client if authentication has failed. May be followed by the server terminating the connection.
- `{spawn, Tag :: any(), link | monitor | none, M::atom(), F::atom(), A::[any()]}`  
`:: tuple()`  
Sent from either party when they wish to start a new process, using the Module, Function, Args (M/F/A) form. Note that the spawn message includes an option to atomically setup a link or monitor so that the calling process can be notified when the remote process terminates.
- `{spawn_res, Tag :: any(), pid() | {error, any()}}`  
`:: tuple()`  
Sent in response to a spawn request. Returns either the PID of the spawned process or an error tuple with a description of what went wrong when spawning the process.
- `{rpc, Tag :: any(), M, F, A}`  
`:: tuple()`  
Sent from either party when they wish to invoke a remote Module/Function.
- `{rpc_res, Tag :: any(), term() | {badrpc, Reason}}`  
`:: tuple()`  
Sent in response to a previous rpc request. Includes the result of executing the Module/Function, or a error tuple indicating what went wrong. Module/Function.
- `{send, from::pid(), to:: (pid() | atom())}`  
`:: tuple()`  
Any message sent asynchronously between two processes. This is the only packet which is followed by “data” message. This permits the SDIST implementation to check the to/from on the packet and validate the sender has permission to send the message before processing the data message.

## 3 Architecture

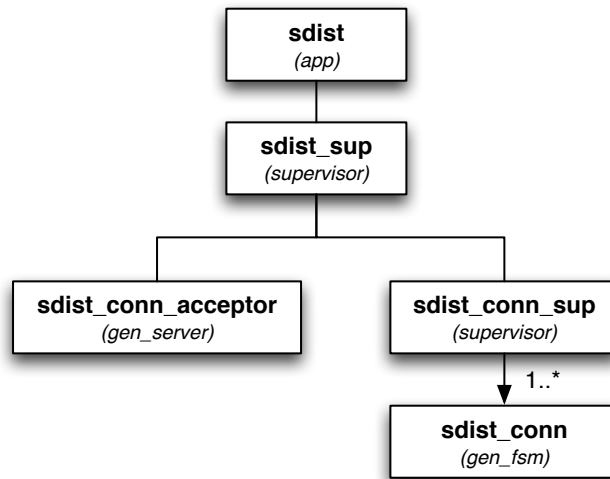
### 3.1 Overview

SDIST is implemented as a standard Erlang application. Initial investigation revealed that the existing distribution mechanism within Erlang is tightly coupled to the VM. Once initial “handshaking” is complete between nodes using the standard protocol, message, monitoring, etc. is all handled by the VM directly – thus it is not possible to filter or otherwise pre-process messages. This is directly incompatible with the design goals of SDIST, since we want to only permit communication with select portions of the running system. For example, SDIST should not permit a remote user to send the halt command to the VM without appropriate ACL checks.

### 3.2 Process Hierarchy

In keeping with standard best practices for Erlang applications, SDIST uses a process hierarchy to provide a robust system. Figure 1 shows this hierarchy of processes.

Figure 1: Process Hierarchy



The top-level supervisor, `sdist_sup`, ensures that the processes responsible for accepting new connections from clients and processing each connection are restarted as necessary in the case of failure. Each inbound connection the node is allocated an independent process (`sdist_conn`), and handled within that scope. This approach ensures that connections can no corrupt each other since their state is fully independent and also permits independent failure of any connection.

### 3.3 Modules

The sdist application is composed of a series of Erlang modules, some of which are instantiated as processes in the process hierarchy.

- `sdist`  
Provides public API to the sdist application. Interacts with the connection supervisor to setup new connection processes and the ACL table process to manage ACLs on specific modules.
- `sdist_sup`  
Application supervisor; uses `one_for_one` restart strategy (that is, any supervised process may die and be restarted independently of the others).
- `sdist_acl`  
Utility module that manages persistent ACL table using Mnesia (Erlang's embedded database). Services that wish to expose an API invoke this module.

The default policy for ACLs is DENY; thus if a given service is not present in the ACL table, access to it will not be permitted.

- `sdist_cred_store`  
Utility module that provides a persistent store of credentials for authentication mechanisms. Used by authentication mechanisms on both client and server to lookup user credentials while performing authentication.
- `sdist_conn_acceptor`  
Worker process that listens for incoming TCP connections and interacts with the connection supervisor to hand off each connection to a dedicated process.
- `sdist_conn_sup`  
Connection supervisor; uses `simple_one_for_one` restart strategy. All connection processes are managed by this supervisor.
- `sdist_conn`  
Worker process that manages a single `sdist` connection between two nodes. This process is implemented as a finite state machine (using `gen_fsm` behaviour). This module interacts with registered mechanism modules (see `sdist_mech`) and the credential store to negotiate the authentication mechanism and perform the authentication. Once authenticated, inbound messages are checked against the ACL table available from `sdist_acl` prior to execution.
- `sdist_client`  
Stand-alone module that provides client access to remote `sdist` instances. Instantiates a process which uses a finite state machine (using `gen_fsm` behaviour), to manage the connection.
- `sdist_mech`  
Behaviour module (aka “interface”) which describes the functions a mechanism must implement in order to authenticate a user. Mechanism implementations extend this behaviour and provide their specific functionality.
- `sdist_mech_anon`  
Implementation of the `sdist_mech` behaviour/interface that provides an implementation of the SASL ANONYMOUS mechanism, per [11]. This permits user to anonymously access services which have the appropriate ACLs configured.
- `sdist_mech_plain`  
Implementation of the `sdist_mech` behaviour/interface that provides an implementation of the SASL PLAIN mechanism, per [10]. This permits a user to access services after plaintext authentication. This mechanism SHOULD only be used after initializing a TLS secured connection.
- `sdist_mech_digest`  
Implementation of the `sdist_mech` behaviour/interface that provides an implementation of a MD5-digested based authentication. This system uses a

server-provided nonce and the user's plaintext password to generate a one-time unique hash. This hash is then transmitted to the server where it is checked against the server's copy of the user credentials. This mechanism has the advantage of not requiring the user's password to be passed in plaintext form over a typical TCP connection.

### 3.4 Implementation Notes

The implementation of `SDIST` was relatively straightforward. Most of the complexity is found in the connection management code. Each connection is allocated a lightweight process which maintains the state for that connection. A finite state machine was used to manage the connection and keep track of what operations were applicable, given the current state of the connection. For example, the system terminates the connection if a user attempts to perform an RPC if the connection has not yet successfully authenticated. Designing the pluggable authentication system also added some minor complexity to the system, but the benefit is significant. The `PLAIN` authentication mechanism was the first implemented and it took about an hour to construct; `ANONYMOUS` and digest-based mechanisms took considerably less time once the framework was in place.

There are a few implementation details that had to be postponed for future work. Foremost among them was support for the `STARTTLS` feature. While the protocol for exercising the feature was implemented in the `SDIST` protocol stack, the Erlang implementation of TLS (which is a part of the VM) has outstanding bugs when it comes to converting a "plain" TCP connection to a TLS-secured connection. Once those bugs are corrected in the VM, it's a minor (2 line) change to `SDIST` to enable support for TLS.

Another implementation problem arose when implementing linking and monitoring of processes. Getting the semantics of the monitoring correct is challenging due to the number of interstitial processes which act as middlemen when providing RPC function calls and message sends. The general expectation would be that if a user starts a linked process, that process (on the remote machine) does not exit if the connection is dropped – or at least not right away. Ideally, the connection would auto-reestablish and the link re-activated. This would likely entail another process (on the remote machine) that starts a timer if the connection process exits and then eventually kills off the linked process if the connection is not reestablished within some time window. However, some users may want stricter implementation, where the linked process is killed immediately. All of these considerations also apply to monitors. Choosing the correct approach requires some deep investigation into the Erlang VM source code to see how it has solved that same problem.

Initial performance review indicates that the `sdist`-based RPC is approximately 10% faster than the default RPC provided by Distributed Erlang. This is a counter-intuitive result, as the Distributed Erlang implementation is primarily C code within the Erlang VM and it does no per-RPC access control checking. However, one

possible explanation for this discrepancy is that the sdist-based RPC invokes the module/function call on the socket process, unlike the original RPC mechanism which dispatches to a secondary process. Regardless, this result suggests that it should be possible to keep sdist on par with the Distributed Erlang system; the cost of the additional authentication should be amortized across the lifetime of a inter-node connection.

## 4 Future Work

There are a number of areas where sdist, particularly the implementation, could benefit from additional work. The protocol should not require any further work. In particular, the implementation support for STARTTLS would be essential if not using the protocol in a fully trusted environment. Having support for ANONYMOUS and DIGEST authentication mitigates this issue somewhat, as the implementation already has value in a low-trust environment, since there are a number of useful services which can be with fine-grained access control. A fully robust deployment of sdist would also require additional safeguards on the External Term Format encoders to ensure that malicious clients could not generate a DoS attack by generating valid requests with large number of atoms (which are stored in a fixed-size symbol table). In addition, the implementation would probably need to support some type of rate-limiting system to ensure that a client could not dominate the connection without regard for other connections.

## 5 Conclusion

This paper has described the sdist protocol and identified how it solves some of the major flaws found in the default Distributed Erlang implementation. sdist provides fine-grained ACL based security which enables individual functions within a module to be exposed in a piecemeal fashion. The pluggable authentication system it provides further ensures that the protocol can support current and future authentication mechanisms, without requiring significant changes; it also addresses the problems that are inherent with a single shared-secret amongst all nodes within a cluster.

## References

- [1] J. Armstrong. *Making Reliable Distributed systems In the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology, Stockholm Sweden, December 2003.
- [2] J. Armstrong. A history of erlang. In *HOPPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–16–26, New York, NY, USA, 2007. ACM.
- [3] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2.
- [4] R. Green. Enhancing security in distributed erlang by integrating access control. Master's thesis, The Royal Institute of Technology, Stockholm Sweden, June 2000.
- [5] [http://www.erlang.org/doc/apps/erts/erl\\_ext\\_dist.html](http://www.erlang.org/doc/apps/erts/erl_ext_dist.html). External term format.
- [6] <http://www.erlang.org>. Erlang.org.
- [7] B. Karlsson. Secure distributed communication in safeerlang. Master's thesis, The Royal Institute of Technology, Stockholm Sweden, October 2000.
- [8] P. Leach and C. Newman. RFC 2831: Using Digest Authentication as a SASL Mechanism.
- [9] A. Melnikov and K. Zeilenga. RFC 4422: Simple Authentication and Security Layer (SASL).
- [10] K. Zeilenga. RFC 4616: The PLAIN Simple Authentication and Security Layer (SASL) Mechanism.
- [11] K. Zeilenga. RFC 4505: Anonymous Simple Authentication and Security Layer (SASL) Mechanism, June 2006.