FINAL REPORT

SUMMER INTERNSHIP

FERMI NATIONAL ACCELERATOR LABORATORY

SCIENTIFIC COMPUTING DIVISION

New flexible, reliable and secure logging channel for distributed workflows

Authors:
Leonardo Lai

Email:
leonardo.lai@live.com

October 11, 2019

Contents

1 Abstract

This document is an overview of the activities that I carried out during my internship at Fermilab between August and September 2019. My task was to develop a new logging channel for glideins within the *GlideinWMS* project, a workload manager for distributed systems. The new channel was required to be reliable, versatile and secure, overcoming most of the issues that affect the current implementation.

2 Scientific computing

2.1 HTC/HPC

Scientific computing exploits computers to solve complex problems that arise from mathematics, physics, chemistry and so on. In this domain, a standard desktop machine is typically not enough, but you need more and more powerful machines to cope with different requirements. Specifically, the expression *High Throughput Computing* (HTC) is used to describe the need of many resources over an extended period of time, whereas *High Performance Computing* (HPC) denotes the same but for short intervals. Scientific experiments are indeed very demanding in terms of computational resources, and the requirements strongly vary between the experiments.
Examples can be easily found in physics. In the *Muon g-2* experiment, which aims to precisely measure the magnetic dipole moment of the muon, the data acquisition system must be able to handle bursts of 18 Gbit/s [6]. The *Dark Energy Survey* (DES) experiment, designed to map the universe and study its galaxies, produces 2.5 TB of image data per night [1]. This is a lot of data, considering that it must be eventually processed. Finally we have the *Large Hadron Collider* (LHC) experiment, whose particle collisions generate the outstanding amount of 1 PB/s. Even though the most of the events get discarded by the trigger, the LHC experiments produce about 15 PB of raw data each year that must be stored, processed, and analyzed [4].
Historically, the request for computing resources for scientific applications has been growing steadily. In the 1960s, the computer that managed to land the first man on the moon was less powerful than today's smartphones by many orders of magnitude. In the 1980s, vertical scaling led to massive supercomputers, nevertheless those were still below modern laptops. It's not until the end of the 20^{th} century that *parallelization* became the paradigm to pursuit to achieve performances that no existing machine could ever reach individually. For instance, the *Earth Simulator* was a supercomputer assembled in Japan out of 640 identical nodes, capable of operating at 35.86 Tflops, used for simulations of the terrestrial atmosphere and oceans.
Today is the era of cloud services and distributed computing. Not only you can have thousands of machines working in parallel, but they can be geographically very far from each other. They are joined together via network to form a single virtual supercomputer. Clear advantages in these solutions are, aside from their commercial applications, the intrinsic scalability (just add more machines), flexibility (diversified hardware/software) and reliability (redundancy is possible).

2.2 HTCondor

Clusters are usually complex systems, hard to manage even for experienced computer engineers, let alone for physicists and other kinds of scientists. While specific expert-level skills are required to setup and configure such a system, the final user is not necessarily expected to be a computer specialist. Therefore you need a mechanism that hides the internal low-level complexity (OS, networking, scheduling, ...), and exposes a simple interface to those who just want to run their code and get the results back. This is done by a *batch system*, or *workload management system*; there are many notable examples, like *HTCondor*, *SLURM*, *TORQUE*, *LSF* [7, 13–15].

HTCondor, developed at the University of Wisconsin-Madison, provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. HTCondor handles submitted jobs placing them into a queue, and also decides where to run them based upon a policy. Eventually, it informs the user upon completion. A unique feature of HTCondor are *ClassAds*, a flexible framework to express job requirements or preferences. HTCondor systems include a single *central manager* node and a pool of several worker machines. The central manager keeps the state of the system, collects updates from the pool and matches jobs to proper resources.

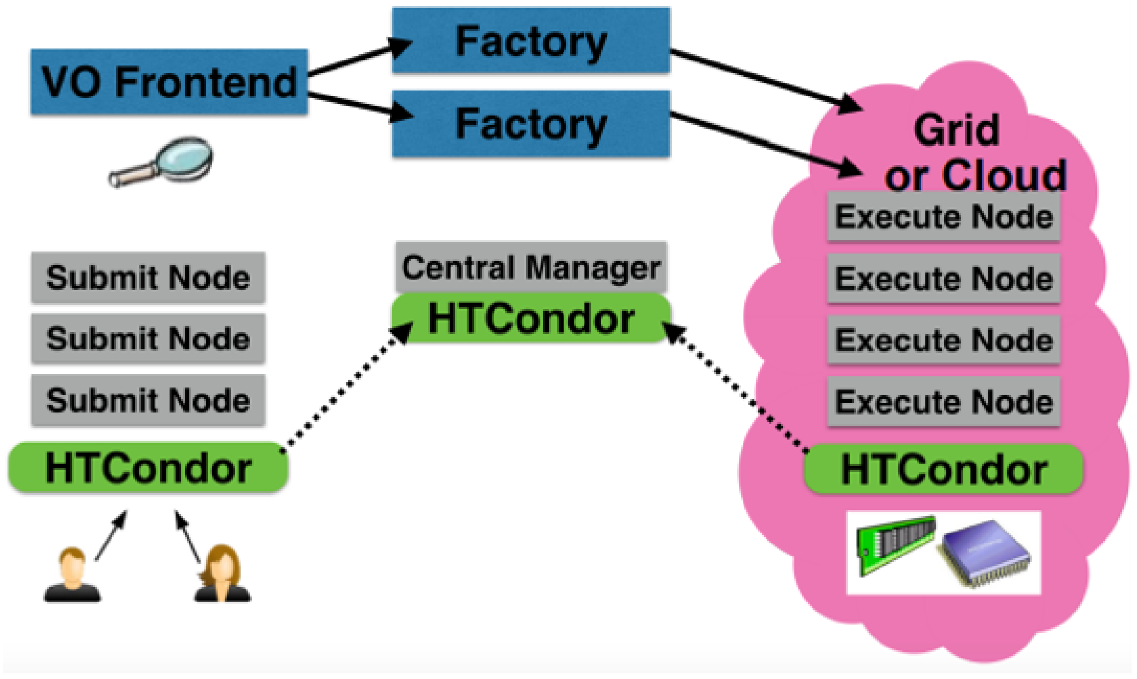


Figure 1: HTCondor architecture

2.3 GlideinWMS

Everything has limits, and HTCondor is no exception. A grid is typically clustered into hundreds of different pools, possibly belonging to different providers, each running its own workload manager. These batch systems are almost never compatible between them natively, so a further abstraction level is needed to form a homogeneous grid-wise pool of resources. A Fermilab project, *GlideinWMS*, addresses this challenge, offering a virtually uniform pool by exploiting the concept of *glidein*, that is a pilot job with a proper configuration. A glidein is submitted as a regular HTCondor job; it performs an initialization routine on the worker node (hardware detection, environment setup, file downloading, error handling, ...) and then, at the end of the procedure, it shows as a resource to the virtual cluster, ready to accept user jobs. The two most important components of the GlideinWMS architecture are the *Frontend* and the *Factory*. The former periodically polls the user pool for queued jobs, and checks whether there is an adequate number of available glideins; the latter is instead responsible for creating and submitting new glideins upon request by the Frontend. The benefits of using GlideinWMS are countless. First of all, it provides support for many different types of sites, including but not limited to Google CE and AWS. The same glidein can be reused to run subsequent jobs from the user pool after one finishes, resulting in increased efficiency. Moreover, glideins provide a kind of fault-tolerance too: if a site does not for some reason, user jobs won't get lost because the failure is first detected by the pilot job. It's important to note that GlideinWMS inherits the *schedd* daemon directly from HTCondor, thus the job submission is exactly the same as before.

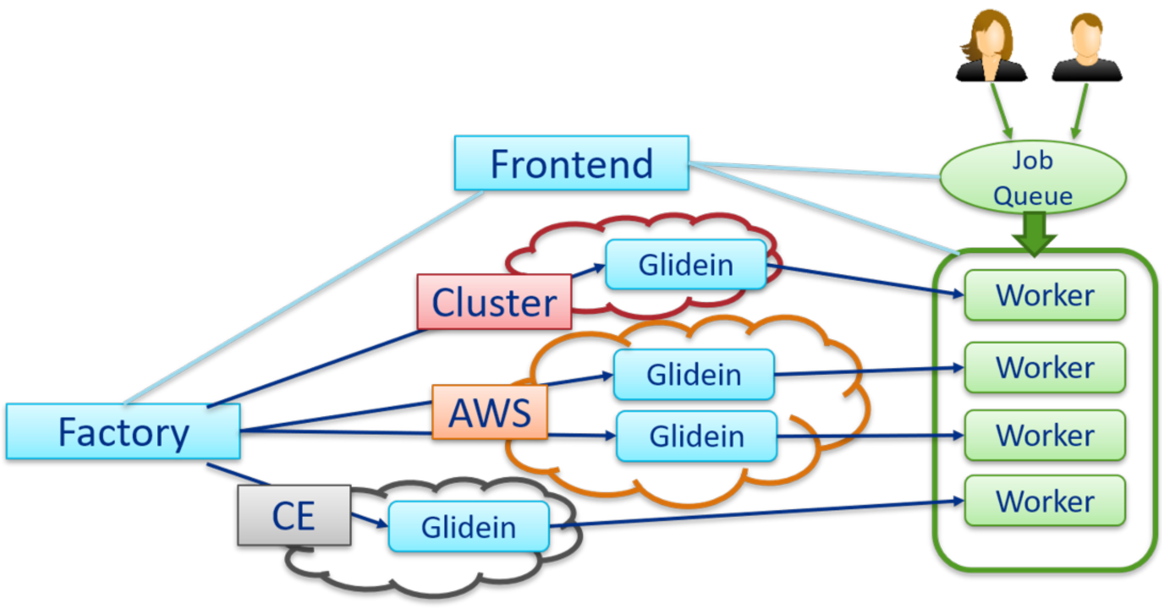


Figure 2: GlideinWMS architecture

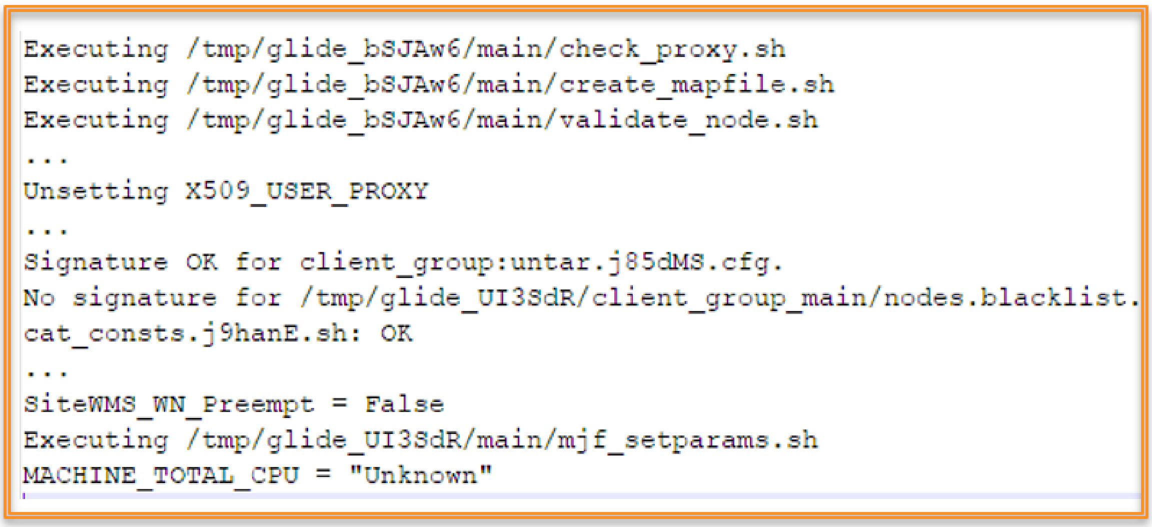
3 GlideinWMS Monitoring

3.1 Overview

Monitoring is a critical task in any complex system; having a detailed view over the state of the system components not only enables you to collect significant statistics and perform fine-grained analysis, but also allows you to precisely locate issues, if any, and eventually fix them. The time spent in debugging activities is in general inversely proportional to the clarity and completeness of the available reports. A solid logging infrastructure is therefore needed to keep both product quality and team productivity above acceptable levels, especially for large projects.
In GlideinWMS, data are collected and handled in different ways. All the logs generated by HTCondor are still available (e.g. *MasterLog*, *CollectorLog*, *StarterLog*), which are particularly useful to troubleshoot problems related to HTCondor configuration, or even low-level issues related to networking and authentication. ClassAds (a structured language used by the entities to express their attributes) messages can also be inspected to view the communications between the objects. Moreover, there are dedicated logs for the Glidein factories, VO frontends and grid entries too.
Unfortunately, debugging the lifecycle of a glidein relies solely on the analysis of the messages print to standard output (*stdout* and standard error (*stderr*). This imposes strong limitations in terms of flexibility and reliability, as shown below.

3.2 Current issues

The content of glidein stdout and stderr is extremely heterogeneous and unstructured. Among the various messages, it is possible to find debug strings, configuration dumps, output of failed commands, and so on. Moreover, the provided information is rather incomplete: very few lines include a timestamp, fewer a severity level, and almost none the subject, that is the name of the script which produced that log entry.



```
Executing /tmp/glide_bSJAw6/main/check_proxy.sh
Executing /tmp/glide_bSJAw6/main/create_mapfile.sh
Executing /tmp/glide_bSJAw6/main/validate_node.sh
...
Unsetting X509_USER_PROXY
...
Signature OK for client_group:untar.j85dMS.cfg.
No signature for /tmp/glide_UI3SdR/client_group_main/nodes.blacklist.
cat_consts.j9hanE.sh: OK
...
SiteWMS_WN_Preempt = False
Executing /tmp/glide_UI3SdR/main/mjf_setparams.sh
MACHINE_TOTAL_CPU = "Unknown"
```

Figure 3: Examples of messages from stdout/stderr.

The fact that stdout and stderr are also exploited as means to return the job results (in XML format) makes the situation even more confusing.
Another issue is the availability of the logs. Stdout and stderr are file descriptors in the worker nodes, so the logs are written there. However, physical access to those machines is generally not possible, because they may belong to various organizations with different security and access policies. In the current implementation, to make stdout and stderr logs available somewhere else too, they are flushed and sent to the Factory once, at the very end of the glidein's life. Problems arise when a glidein terminates prematurely. In UNIX-like systems, every process has its own stdout and stderr FD; these exist for as long as the respective process is alive, but are permanently destroyed thereafter. Hence, if a process dies abruptly (e.g. when killed with a signal) then its stdout/err are immediately freed as well. A glidein is nothing a process, so it suffers the same behavior: terminating it from the outside (with a signal) causes the immediate deletion of its stdout and stderr logs, which will never make their way to the Factory and become available. This is a major limitation in terms of troubleshooting, since these logs could be very useful to analyze the glidein state right before its death, especially if anomalous.
Multi-threading creates problems too. A glidein may either execute in a single thread or spawn multiple ones. Another source of possibly concurrent threads are the cron jobs. It is well known that race conditions can occur when two or more entities interact at the same time with the same resource, in this case the stdout/err. A possible scenario involves two threads attempting to write to stdout simultaneously: the result is unpredictable, and not necessarily the same as if the threads were executed one after the other. Not only this affects the quality of the log in terms of readability, but also represents a major difficulty for automatic tools that parse and analyze it.
Finally, the current system does not provide any mechanism to forward/broadcast the logs to multiple nodes other than the Factory. In practice, this means that one must access the Factory machine to view the logs, instead of reading them in a dedicated log server.

4 New logging system

4.1 Overview

In order to overcome all the aforementioned issues, a new independent logging channel is added to GlideinWMS. It is specifically designed to:

- provide complete information in a structured format
- work correctly even in multithreaded scenarios
- be independent from the process
- make the logs available wherever needed
- be secure
- be easy to use

4.2 Format

The new logging channel adheres to the *JSON* (JavaScript Object Notation) format. The choice of JSON against other options is motivated by practical considerations. With respect to XML, it is more compact and definitely more readable. YAML could be an alternative too: despite it being less verbose, JSON is easier to parse and sometimes faster to deserialize.
Every glidein has its own log file. Each entry of the log is represented by a JSON object, containing useful information about the recorded event. The core is the ”content” attribute, which stores the log line as a string. Actually, depending on the ”type” field, the content may contain either a standard plain-text message or the body of a chosen file. The attributes are:

– invoker: the name of the script that logged the event

– pid: ID of the process that logged the event

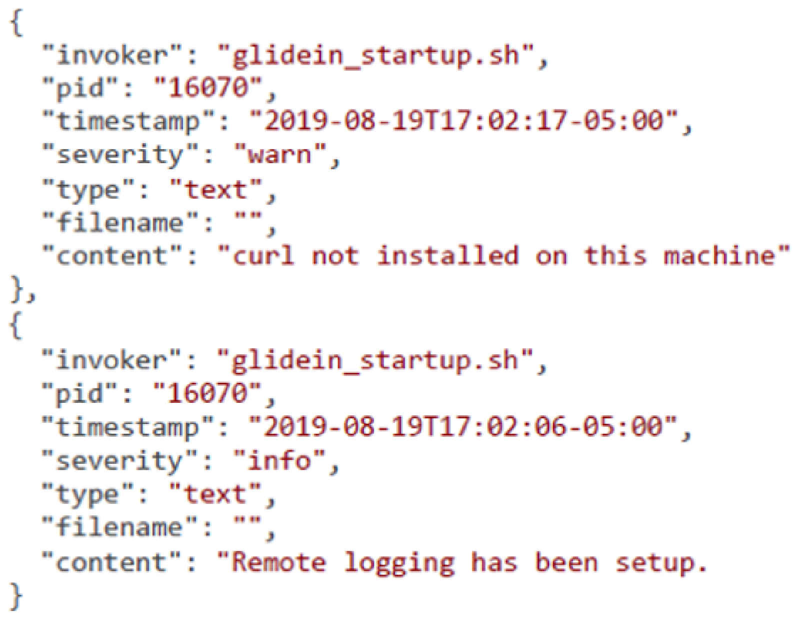
– timestamp: instant when the event was recorded

– severity: severity level of the logged information

– type: determines if the content is a message (”text”) or a file (”file”).

– filename: filename if type is ”file”, empty otherwise.

– content: string message if type is ”text”, or the content of the specified file encoded in base64 if type is ”file” and compressed. Encoding helps sanitizing the input.



```
{
  "invoker": "glidein_startup.sh",
  "pid": "16070",
  "timestamp": "2019-08-19T17:02:17-05:00",
  "severity": "warn",
  "type": "text",
  "filename": "",
  "content": "curl not installed on this machine"
},
{
  "invoker": "glidein_startup.sh",
  "pid": "16070",
  "timestamp": "2019-08-19T17:02:06-05:00",
  "severity": "info",
  "type": "text",
  "filename": "",
  "content": "Remote logging has been setup.
}
```

Figure 4: A couple of log entries with the new JSON-based format

The whole log is a JSON array containing all the event objects sorted by timestamp. Additionally, another object (metadata) is prepended to the list: its attributes provide information about the log subject, that is the glidein, including the names of the Factory, Frontend, Group, Entry, and many others.

4.3 Concurrency

As previously mentioned, multiple processes/threads may run concurrently within the context of the same glidein. In this case, uncoordinated simultaneous write accesses to the log file, which is shared among these threads, would interfere with each other and jeopardize the log integrity.
A possible solution is to use of locking mechanisms (mutex, semaphores, monitors). A properly designed locking scheme would indeed ensure the correctness of the system, though affecting non-functional properties like performance. When acquiring a lock, a process must wait if another holds it; such wait can be detrimental for the system efficiency, especially if the shared resource is frequently needed and many threads compete for it. Moreover, locking protocols are notoriously delicate and easy to mess up when implementing them from scratch; the consequences may be unpredictable and tricky to figure out, ranging from minor malfunctions to deadlocks. Due to these reasons, a lock-free solution would be preferable.
Another possibility is *sharding*: the idea is to write every log entry to a separate file, and then merge them to a single log. This way the threads cannot interfere because they do not write to the same location when logging a new event. However, you still have the problem of merging (*coalescing*): who should it, how and when? If you choose to merge the shards only at the end of the glidein's life, then the log file will be written by the last thread without race conditions, but the log itself won't be available until very late: this is a strong limitation in terms of flexibility. On the other side, if you let any thread perform the merging at any time, then the problem of having multiple access to the same resource (the log file) rises again. Furthermore, what could happen if a thread starts the merging procedure meanwhile another is writing a shard?
The real solution is a hybrid of the previous ones, based on three principles:

- sharding
- minimal use of locking (hidden)
- isolation between operations

The whole logging process takes places in four distinct directories: ”logs”, ”shards”, ”creating” and ”coalescing”. When a process produces a new log entry, what actually happens is the creation of a shard file in the ”creating” directory, whose name replicates info of the body (e.g. timestamp, severity, ...) to speed up post-processing operations like sorting. The ”creating” directory protects the files from other accesses while they're being created; When the shard is fully written, it is moved to the `shards` folder, where it becomes ready for subsequent operations like merging. When a process wants to join all the shards, it first checks whether the `coalescing` directory already exists: if so, all the files in `shards` are moved to `coalescing`, along with a copy of the current log from `logs`; otherwise, the coalescing request is dropped. All the shards in `coalescing` are then concatenated to the previous log in chronological order, to form a new log file. The latter replaces the older one in `logs`, and the `coalescing` folder is forcefully deleted together with its content. This pattern may look rather intricate, but it ensures that two processes never operate on the same resources concurrently, indeed there are different folders for writing, merging and storing. Although there is no explicit use of locks, the scheme is correct only under the condition of atomic ”move” and ”create directory” operations. Luckily, this property holds for the Unix commands `mv` and `mkdir`, at least for local file systems. Their implementation actually uses locks at kernel level, but it is no problem as long as lock/unlock operations are sporadic and rarely block. Of course, we assume that the OS primitives are bug free.

4.4 Log forwarding

The log is generated in the worker node, a location which is far from being the ideal place where to have it. Typically these machines belong to different remote organizations, and accessing them may not be even possible due to security policies. An option is to forward the log to the glidein's Factory: this can be easily done by appending the log to the stderr, which is flushed anyway to the Factory when the glidein terminates. It is a good solution, yet not so flexible. Other problems persist too, like the missing logs for killed glideins.
Another possibility is sending them to one or more HTTP servers. Their URLs can be specified in the Factory configuration file with the attribute `log_recipients`:

```
<attr const="False" glidein_publish="False" publish="False"
name="LOG_RECIPIENTS_FACTORY" parameter="True"
job_publish="True" type="string" value="url.com/log" />
```

If multiple servers are present, their URLs must be separated by white space. In the future, these addresses will be configurable from the Frontend too.
Analogously, the *LOG_RECIPIENTS_FACTORY* attribute can be individually set for any entry, in its own *attr* tag.
When a glidein requests to send its log to a remote server, it also includes a dump of the produced stdout and stderr; three HTTP PUT requests are sent out, one for each file (stdout, stderr, log). The server usually replies with a ”201 Created” in case of success, or another HTTP code if something else occurred. The full log is sent anyway whenever the glidein is about to end, regardless of whether the termination was spontaneous or forced by external processes.

4.5 Security

Logs are sent from the entry nodes to the log server over the Internet, which is an inherently insecure channel unless to adopt proper cryptography protocols. Some of the potential threats to account in the design of the system are:

- Logs being sniffed and read by unauthorized parties (*eavesdropping*)
- Entities posting logs to the server without authorization
- Malicious agents pretending to be someone else when posting logs to the server (*impersonation, forgery*)
- Attackers being able to tamper communication messages (*malleability*)
- Authentication secret information being stolen by inspecting the packets *(identity theft)*

Cybersecurity is a complex matter, full of pitfalls and where vulnerabilities always find their way to sneak in. Thus it is generally recommended to use consolidated standard schemes rather than inventing new protocols from scratch, and we follow this approach hereafter.

4.5.1 Assumptions

Some assumptions were identified before designing the security scheme:

- The GlideinWMS software is secure, and the scripts provided by the VO Frontend are secure as well.
- The hosts are responsible for keeping the *secret keys* safe from malicious agents, making them unlikely to be stolen. If an intruder ever steals a secret key, the security is assumed to be compromised for all the entities relying on it. In such circumstance, it is desirable to mitigate the negative consequences as much as possible, although not strictly required.
- Log servers can be located anywhere; it may hosted on the same machine of a Factory, a Frontend or simply be standalone.

4.5.2 Technologies

The following cybersecurity concepts are at the heart of the implemented scheme.

- *Digital Certificate*: electronic document used to prove the ownership of a public key. It contains information about the identity of the owner, and is generally issued by a trusted Certification Authority (CA).

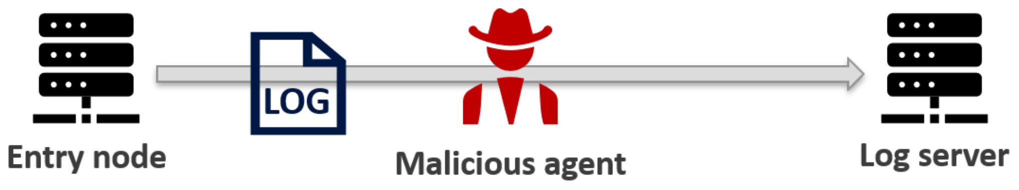


Figure 5: A ”man in the middle” in the network between the glidein and the log server can intercept and manipulate the packet if no security mechanism is enforced.

- *SSL (Secure Sockets Layer) [5]*: cryptographic protocols designed to provide privacy and data integrity between hosts in a network.
- *TLS (Transport Layer Security) [2]*: the modern successor of (deprecated) SSL.
- *JWT (JSON Web Token) [8]*: a small JSON object used to safely exchange claims between two parties. The header defines which protocol to use, the payload contains the actual claims, and the signature is used to verify the token itself. A secret key is used for both generating and verifying it.
- *Symmetric key cryptography*: protocols where two entities share a secret, that is used to encrypt/decrypt or sign/verify their messages. The secret must never be revealed to third parties.
- *Asymmetric cryptography*: protocols where the entities own pairs of corresponding private and public keys. The private one must never be revealed to others, and it is used to encrypt or sign documents. The other key is instead of public domain, and used to decrypt messages or verify signatures.

4.5.3 Security model

When a glidein wants to send a log to any server, it must include a JWT in the HTTP Header field. The Factory issues (and signs) the token and transfers it to the glidein via HTCondor secure transfer.
At *reconfig* time, the Factory predetermines the list of tokens to be eventually transferred to each glidein, on the basis of the configuration files. These are only names, though: the tokens are effectively created later, when the Factory service starts.
The triplet {Factory}-{entry}-{recipient} uniquely identifies a token, provided that name duplicates do not exist.

4.5.4 Token structure

The header defines the algorithm to use for signing. Here we use HS256, that is HMAC with SHA256 [3, 9], which requires a shared secret between the issuer and the audience. A public key protocol (RSA [12]) would even work better, but the implementation requires Python 2.7 and there are still SL6 machines supporting Python 2.6 only. The payload contains the following fields:

- sub: Subject of the token; it is the entry name
- iss: Issuer of the token; can be a Factory or a Frontend.
- aud: Audience; it is the recipient address
- iat: Timestamp of issuance
- exp: Expiration timestamp
- nbf: First timestamp of validity

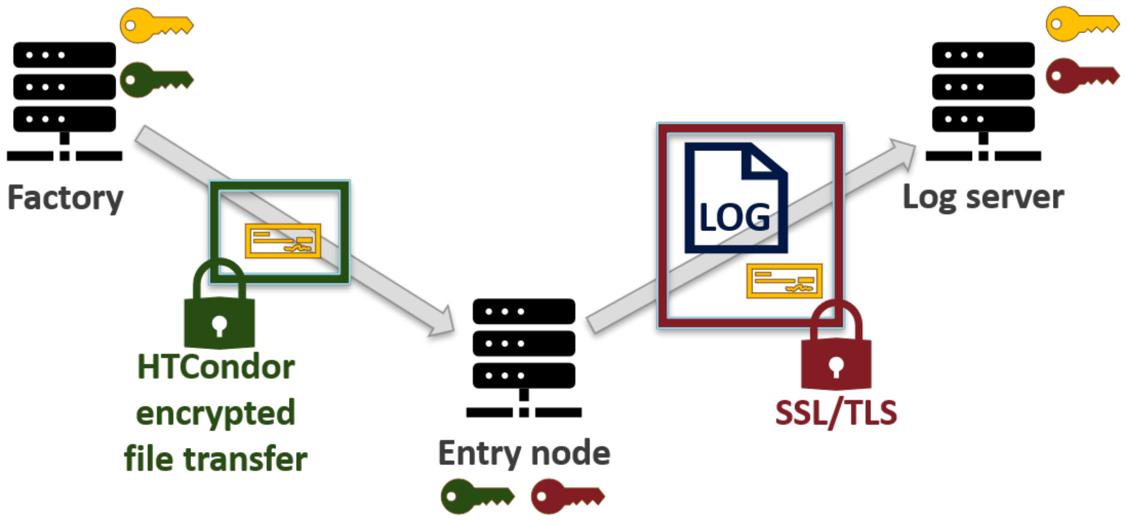


Figure 6: Overview of the security model for the logging channel: a token (yellow) is issued by the Factory and used by the glidein in the entry node to authenticate itself to the log server. All the exchanged messages must be encrypted; in the figure symmetric cryptography is adopted (corresponding keys are of the same color), but public-private schemes are possible as well. HTCondor natively offers options to transfer files securely.

Given that the lifetime of a glidein never exceeds few days, the tokens should remain valid for approximately one week, after which they must be renewed. The issuer is responsible for scheduling a periodic update of its tokens, e.g. setting a cron job up.

4.5.5 Factory token organization

The Factory stores the token in a directory called *server-credentials*. There you have a sub-folder for each configured entry, named {*entry_name*}, which in turn contains a sub-directory *tokens*. Going deeper, there is one more folder for each recipient enabled for that entry. Their names are derived from the server URLs, just encoded to get rid of slashes and globbing characters. Finally, inside these directories, you find the tokens: if the recipient is set in *frontend.xml*[1], then the name is the Frontend's one, otherwise it is equal to *default.jwt* and is signed by the Factory. For instance:

```
# ls /var/lib/gwms-factory/server-credentials/entry_ITB_
FC_CE2/tokens/https%3A%2F%2Ffermicloud152.fnal.gov%0D%0A/

default.jwt  voFrontend1.jwt  vofrontend2.jwt
```

4.5.6 Copying tokens from Factory to glidein

When a glidein is about to be spawned, the Factory must transfer the correct tokens to the machine hosting that glidein. Since the token are sensitive files, their trans-

[1]frontend token generation is not fully supported yet

mission requires the use of a secure encrypted channel, to prevent them from being intercepted and stolen by third parties. HTCondor *condor_submit* provides support for secure file transfers with the directives *transfer_input_files* and *encrypt_input_files*. The GlideinWMS Factory just needs to specify a list of files, and they're automatically sent to HTCondor workspace in the entry node.
In theory, the selected files should depend on the Frontend (if it specified a custom token)[1], the submit entry and the configured log recipients. Unfortunately, the GlideinWMS code architecture makes it hard to customize the set as function of the Frontend and recipients:

> ☞ In GlideinWMS, the list of files is currently generated statically in *cgWConsts.py* at Factory reconfig time, hence there is no way to dynamically choose what files to send on the basis of the glidein's Frontend. Moreover, since the Frontend can change the log recipients on its own, without triggering a Factory reconfig, not even these can be used to tailor the files list. A workaround is to transfer multiple tokens (all those associated to the entry) inside a statically-named archive to the glidein, with the latter being responsible for immediately discarding the unneeded ones at startup (*glidein_startup.sh*). The security still holds, provided that the token cleanup is correctly executed and that files never leak out of the glidein workspace (i.e. glideins are isolated).

The tokens are archived as tarfile and zipped (*tokens.tgz*) before being sent, along with a descriptor file (*url_dirs.desc*) that helps the glidein figuring out the association between recipients URLs and the directory names. On the receiving side, the glidein, which is aware of its Frontend and Factory names, preserves only the tokens that are strictly necessary to talk with the log servers, and forcefully deletes all the others.

4.5.7 Glidein using the token

From the glidein's perspective, it is very easy to use the received tokens for authentication. Given the recipient, it just needs to select the corresponding token, include it in the HTTP message header and send the request. No further action is needed.

4.5.8 Server-side verification

When a log server receives a request, it first checks if the token is present. If not: end of the story. Next it reads the issuer field, to figure out what secret key should be selected for verification. The key must be already present in the server, either manually deposited or automatically exchanged with the issuer using a protocol like Diffie-Hellman[2]. Note that the token hasn't been verified yet, so any information read should not be really trusted. Anyway, the worse that could happen is an entity that intercepts a token, strips the signature, replaces the issuer field with its name and finally re-signs it: this is basically pointless and not harmful at all.

[2]Automatic key exchange not supported yet

The server then verifies the token using the secret associated to the issuer, validates the other fields (especially the expiration time) and, if everything goes well, the http request is handled.

4.5.9 Other considerations

The system does not provide explicit countermeasures to DoS (*Denial of Service*) attacks; however, these can be mitigated by a log server in many ways, including firewall rules, Frontend hardware or more sophisticated tools.
The fact that more tokens than necessary are forwarded to a glidein, due to technical constraints, represents a weak point of the system from two points of view: security and scalability. Agreed that sensitive data should travel as little as possible, the system security relies on the correctness of *glidein_startup.sh* discarding the extra tokens: this functionality deserves particular attention, and should be carefully tested. Second, sending more tokens implies an increase in the network traffic volume for the Factory. Given that the average token size is approximately 300 bytes, the overhead is still negligible until the number of Frontends and recipients remains fairly limited and bounded to a few dozen. To scale beyond this limit, a solution could exploit a message queue (*Kafka* [10], *RabbitMQ* [11], ...) that receives messages from the glideins and forwards them to log servers based on application level information.

4.6 API

The new logging channel is pretty straightforward to use, thanks to a neat API. All the functions are stored in the file `logging_utils.source` under `creation/web_base`.

4.6.1 Initialization

The log system initialization is a two-phase process. The first step (`log_init` function) is executed once by `glidein_startup.sh`, and results in the generation of the tokens, creation of some log folders and duplication of the stdout/err streams. The second half (`log_setup`), instead, must be executed by every script that wants to use the logging utilities. Its purpose is just to retrieve some environment variables from the glidein configuration. Initialization is mandatory to make the other logging primitives work correctly; if skipped, a warning message is generated when attempting to log or merge events.

```
log_init <glidein_uuid> <relative_basepath>

    glidein_uuid      -> glidein's Universally Unique ID
    relative_basepath -> path where to create the logs dir
```

```
log_setup <glidein_config>

    glidein_config    -> name of the glidein config file
```

4.6.2 Writing

A script can add a new entry to the log by invoking the function `log_write` with proper arguments. The function may fail in case of configuration problems.

```
log_init <invoker> <type> <content/path> <severity>

    invoker      -> name of the script that is logging
    type         -> "text" (string) or "file"
    content/path -> string message or file body
    severity     -> severity level of the logged information
```

4.6.3 Merging

To merge the shards, simply call `log_coalesce_shards` without arguments. Anyway, is generally redundant to do so, since this function is also invoked inside `send_logs_to_remote` (see below). The operation may fail in case of configuration problems.

```
log_coalesce_shards
```

4.6.4 Remote forwarding

Logs can be forwarded to the configured remote servers by invoking the function `send_logs_to_remote`. At the beginning, it will call `log_coalesce_shards` too. The operation may fail in case of configuration problems.

```
send_logs_to_remote
```

5 Future steps

5.1 Frontend token generation

The Frontend should be allowed to set the attribute *LOG_RECIPIENTS_CLIENT* in *frontend.xml*, which extends the list of log servers with new elements. In other words, if the Frontend defines *LOG_RECIPIENTS_CLIENT*, all the glideins related to that Frontend will send logs to those servers too, authenticating with the tokens provided (and signed) by the Frontend: an additional attribute, *LOG_CLIENT_TOKENS*, should contain the file names of such tokens. Of course, *LOG_RECIPIENTS* and *LOG_TOKENS* must be of the same length. In case of overlapping entries between Factory and Frontend (i.e. the same log server is specified by both), the Frontend's one is preferred and overrides the Factory's attribute.

5.2 Token renewal

Tokens remain valid only for a limited period, after which they expire. This behavior is motivated by security considerations: if a token is stolen, the attacker's bad actions are bounded to a finite interval of time. On the other side, it also means that tokens must be periodically renewed or the system will eventually stop working. The most straightforward to do so would be a cron job on the issuer side (Factory). which executes with a frequency that is neither too low (don't let tokens expire) nor too high (avoid excessive overhead). In any case, the period must comply with the following formula:

$$T_{token} > T_{renewal} + T_{glidein}$$

where T_{token} is the token validity interval when issued, $T_{renewal}$ is the renewal period and $T_{glidein}$ is the maximum expected lifespan for a glidein. It ensures that a glidein always receives a token that will be valid for the whole of its life (from spawn to death), and will not expire in the meanwhile.

6 Conclusions

The new logging channel was designed and developed in two months and, although some features are only sketches, most of the mechanisms have been tested with success under different scenarios. The system is not ready for production yet, since a robust server-side log infrastructure is needed too, but the proof of concept is already there. From a personal point of view, it was an outstanding opportunity to work in a professional team, get involved in the workflow and decision processes, learn new technologies and, last but not least, visit another country and enjoy its culture.

7 Acknowledgements

This work could not be possible without the help of my colleagues and other people. I want to especially thank my supervisor Marco Mambelli, who constantly guided me with invaluable dedication; Lorena Lobato Pardavila, whose enthusiasm and technical tips have greatly benefited my internship; Dennis Box, Vito Di Benedetto, Marco Mascheroni, Jeff Dost, who also contributed to my work with their expertise; Giorgio Bellettini, Simone Donati, Emanuela Barzi, for their outstanding effort organizing such a good quality internship program every year; Kappatolia Sherman, Valery Stanley, Rosa P Foote, who assisted me in obtaining a VISA and accommodation; the other italian interns, with whom I have shared many beautiful moments; Fermilab and my home institution, Sant'Anna School of Advanced Studies, which gave me funds to allow a positive stay in the US for two months.

8 Appendices

This sections collects a couple of other activities that I carried out in GlideinWMS, which are not strictly related to the main task (log channel), but nevertheless worth of mention and description.

Appendix A Self-extracting scripts

Some utility functions must be shared between the bash scripts. There are two typical approaches to do so:

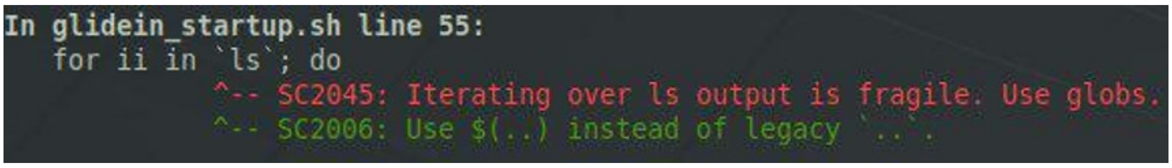
- write the shared code in a standalone file, then the other scripts will source it
  - *Pro*: clean organization
  - *Cons*: functions available only after moving the files to the grid nodes; slightly increased transfer overhead
- Heredoc: put the shared code inside a string; the main script, on execution, writes that string to a file that can be sourced by the other scripts requiring it
  - *Pro*: constant number of files
  - *Cons*: messy organization; need to escape special characters inside the string; editor syntax highlighting spoiled; hard to test

There exists another smart approach which combines the best of the two worlds: self extracting scripts. The idea is to develop the utility code in standalone files, but instead of directly transferring them they are archived, compressed and concatenated to the main script; only the latter is actually sent. On the receiving side, as soon as the main script executes, it automatically strips the utility section from its own file, unzips it and finally shares the contained code by writing it into new files.

Appendix B ShellCheck CI

Testing is an important step in the software lifecycle: it ensures that the program works as expected before the code is deployed to production servers. Depending on the specific application, software bugs may cause a component or the entire system to fail, with consequences ranging from negligible up to catastrophic. Therefore it is very critical to have a solid testing infrastructure in a large project. As such, GlideinWMS exploits *Jenkins*, a server for continuous integration that allows the periodic execution of automated tests.
Static analysis is a class of testing techniques that inspect the source code looking for potential faults, yet without running it. One of these tools is *ShellCheck*, a linter for bash scripts, that is an utility which analyzes the code to flag programming errors, bugs, stylistic errors, and suspicious constructs. I have integrated ShellCheck verification of GlideinWMS bash scripts in Jenkins CI, so that we can monitor better

the presence (and number) of issues in the project now, and possibly improve the overall quality of the code.



```
In glidein_startup.sh line 55:
   for ii in `ls`; do
             ^-- SC2045: Iterating over ls output is fragile. Use globs.
             ^-- SC2006: Use $(..) instead of legacy `..`.
```

Figure 7: An example of warning messages generated by ShellCheck.

References

[1] TMC Abbott et al. “The Dark Energy Survey: Data Release 1”. In: *Astrophysical Journal: Supplement Series* 239 (Nov. 2018).

[2] Tim Dierks and Eric Rescorla. “RFC 5246-the transport layer security (TLS) protocol version 1.2”. In: *Internet Engineering Task Force* (2008).

[3] D Eastlake III and T Hansen. “RFC 4634-US Secure Hash Algorithms (SHA and HMAC-SHA)”. In: *Motorola Labs and AT &T Labs* (2006).

[4] Lyndon R Evans. *The Large Hadron Collider: a marvel of technology*. EPFL Press, 2009.

[5] Alan Freier, Philip Karlton, and Paul Kocher. “The secure sockets layer (SSL) protocol version 3.0”. In: (2011).

[6] Wes Gohn. “Data Acquisition for the New Muon g-2 Experiment at Fermilab”. In: *Journal of Physics: Conference Series* 664 (June 2015). DOI: 10.1088/1742-6596/664/8/082014.

[7] *IBM Spectrum LSF*. https://www.ibm.com/us-en/marketplace/hpc-workload-management.

[8] M Jones, J Bradley, and N Sakimura. “Rfc 7519: Json web token (jwt)”. In: *Date of retrieval* 5 (2015), p. 2017.

[9] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. “RFC 2104: HMAC: Keyed-hashing for message authentication”. In: *Internet Engineering Task Force* 252 (1997).

[10] Jay Kreps, Neha Narkhede, Jun Rao, et al. “Kafka: A distributed messaging system for log processing”. In: *Proceedings of the NetDB*. 2011, pp. 1–7.

[11] *RabbitMQ*. https://www.rabbitmq.com.

[12] Ronald L Rivest, Adi Shamir, and Leonard M Adleman. “Cryptographic communications system and method”. In: *US Patent* 4405829 (1983).

[13] Todd Tannenbaum et al. “Condor – A Distributed Job Scheduler”. In: *Beowulf Cluster Computing with Linux*. Ed. by Thomas Sterling. MIT Press, Oct. 2001.

[14] *TORQUE Resource Manager*. `http://www.adaptivecomputing.com/products/torque/`. Accessed: 2010-09-30.

[15] Andy B Yoo, Morris A Jette, and Mark Grondona. “Slurm: Simple linux utility for resource management”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.