Ahmed Hameed
Danny Little
Zane Bookbinder
May 16, 2022

Global PassStore (GPS)

Abstract

Global PassStore (GPS) is a distributed password management system that securely stores passwords by splitting them into chunks and storing each piece on a different server node. Our system employs functional homogeneity for scalability and global access, clusters for manageability and speed, and replication for fault-tolerance.

Introduction

The problem we seek to solve with this system is password security. In particular, we recognize a vulnerability with storing password data on a single machine, even if that data is completely encrypted. If that machine is hacked by a malicious actor or compromised due to internal bugs or errors, then users' sensitive and private data is at risk of being leaked or exposed. Even though a password can be encrypted with the user's private key, it can be easily decrypted and stolen. To avoid these security risks, we propose splitting passwords into 'chunks' and storing each chunk on different machines located across the globe. With this design, storing and retrieving passwords becomes more complicated and computationally expensive; however, to steal a user's data, an attacker would have to obtain access to multiple geographically-distributed machines, and find the correct pieces. In other words, hacking a single machine reveals very little because the attacker will only have a piece of the password.

Related password management systems include RSA and iCloud Keychain. As of 2012, RSA employed a system called "distributed credential protection," which involves breaking a password into many small pieces and distributing those pieces across two machines.[1] RSA is also able to determine password accuracy without assembling the pieces, ensuring that the whole password is never available for an attacker to steal. While both this solution and GPS don't account for passwords being stolen from the end-user, they ensure that the end-user is the only place where obtaining a full password is possible. Our solution builds on RSA by allowing for more than two password chunks and by distributing these chunks globally, adding an element of physical security.

Another standard password management system is iCloud Keychain, which uses 256-bit AES end-to-end encryption and two-factor authentication. This model tries to ensure that passwords can't be decrypted by anyone other than the user, but it still relies on the security of the user's password. In other words, obtaining someone's AppleId password gives direct access to their iCloud Keychain (two-factor authentication adds an extra layer of complexity to this).

We provide a breakdown of the following sections. In the Design section, we discuss our priorities when designing GPS and how those priorities led us to choose functional homogeneity, location-based clusters, and smart clients. In the Implementation section, we discuss some relevant data structures and walk through the propagation algorithm we used to notify all nodes of a new password. In our evaluation section, we show the results from four tests of GPS under different conditions, all of which show its scalability and fault tolerance. Finally, in our Conclusion, we discuss where our system succeeds and next steps that would improve upon GPS.

---

[1] https://www.technologyreview.com/2012/10/09/183378/to-keep-passwords-safe-from-hackers-just-break-them-into-bits/

Design

As with any distributed system, our main priority was scalability. A password manager is of limited use if it can only handle a small number of passwords or client requests, can only run on a small number of servers, or if performance significantly decreases at scale. Additionally, we sought to obtain high availability and fault tolerance so that our system is always accessible, even if nodes fail or GPS is under heavy load. We considered performance to be our last main priority (with some important caveats). Given the nature of a password manager, users will spend the majority of their time retrieving the same passwords and a minimal amount of time registering new passwords, and altering or deleting old ones. This means that reads from GPS should be relatively quick as they will be frequent, while writes to the system have more leniency in terms of expected performance, and those operations can afford to be a bit slower. Every distributed system has tradeoffs, and in our case, we traded simplicity and some performance for scalability, availability, and core functionality (distributed password chunks).

To promote scalability, we decided to use functional homogeneity, meaning that each server node can respond to register, search, and deletion requests from clients directly without contacting a master server. This choice increases our system's complexity but it greatly increases the availability of GPS as every server node is identical in its functions and should be able to respond to any kind of client request. Furthermore, homogeneity prevents the single-point-of-failure and bottleneck that a master server would cause, which allows for improved scalability of our system.

Continuing, we use node clusters to limit network traffic and help pass fewer messages around the network. We designed the system so that the password chunks for a single password

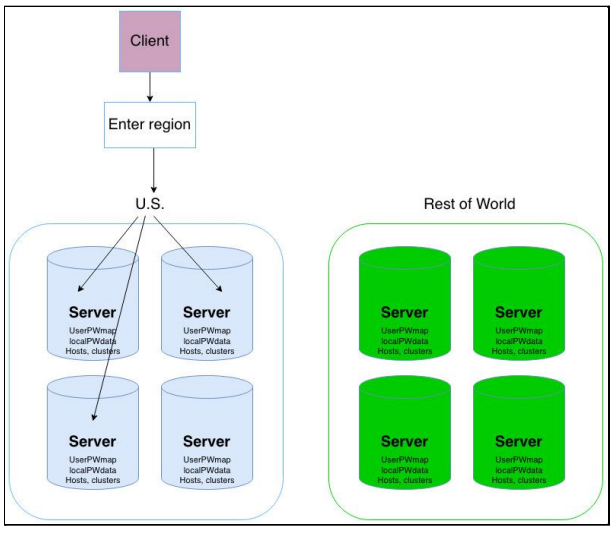
are stored on one cluster and are replicated on a different cluster. In GPS, node clusters are collections of nodes that are close to one another with proximity defined in terms of geographic distance. The purpose for this is to allow for faster and more scalable update propagation and password retrieval operations, as nodes within a cluster should be optimized for passing messages between each other very quickly. Clusters also increase the manageability of our system, as we can command individual nodes to update the rest of their neighbors within each cluster. This design is preferred to having a single node reach out to every other node in the case of an update, as that presents a scalability problem.

GPS employs smart clients for load balancing so that clients can connect to and be serviced by the closest and most available server node. Rather than connecting to a random server, the client program polls 10 nodes in the nearest cluster (picked by the user manually) and connects to the server with the lowest ping time. This algorithm accounts for physical distance and load, as a highly-loaded server would take longer to respond (and therefore would be less likely to be chosen).

Lastly, the GPS system implements replication in other clusters for all password chunks. This provides a greater degree of system fault tolerance as replicas are used to recreate local password data in the case of node faults. Similarly, we provide node joining functionality that automatically detects the optimal cluster and adds a new node into the system.

Implementation

The GPS system for password management is fully programmed in Python and executed using bash scripts. Due to the time constraints of this



project, we found the brevity and simplicity of Python to be very desirable despite the tradeoffs in speed. The following mini-sections discuss the various data structures and data propagation methods that support our system implementation.

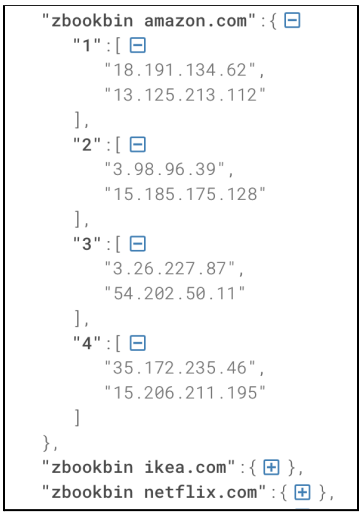
Core Data Structures:

To facilitate functional homogeneity, GPS relies on some key data structures for tracking user data and password chunks. The actual password data for a given user is stored locally within individual server nodes in one cluster and replicated on another cluster. The mapping of user password chunks to server nodes is a shared data structure amongst all the nodes in every cluster, which necessitates update propagation operations to keep this mapping current and accurate. Put in other words, in order for global availability and user access from any server node: all servers need up-to-date information on where user password chunks are stored.

Local Password Data

The local password data structure is a dictionary that maps user account data to the password chunk string that is stored on that server node. Figure 2a depicts the formatting of keys: "[username] [site name][password chunk order number]" and values are the actual strings representing that password piece.

Fig. 2a: An example of the local password data structure in GPS that maps user data to password chunks.



User Password Map

The user password map is a dictionary that tracks where password chunks are stored in the GPS system. It maps username and website keys to another dictionary of password chunk order numbers and the

Fig. 2b: An example of the user password map structure in GPS where user accounts are mapped to password chunks and the server nodes that manage those chunks.

IP address of the server node that is storing that password chunk. Figure 2b shows that there are multiple IP addresses stored for a given chunk. This is due to the replication of all password chunks. This mapping is updated when a client registers a new user account and password, a password chunk is replicated, when clients delete passwords, or when a node fails and its data must be located and re-replicated.

Cluster-based Storage and Propagation:

As discussed previously in the design section, GPS uses clusters to organize groups of nodes based on their proximity in terms of network latency. Currently, these clusters are configured manually but we hope to create features for automatic cluster formation based on low network latency measured through ping time. So the current system includes an Americas cluster and a Rest of World cluster. When registering a password on the Americas cluster, only one machine in the Rest of World cluster needs to be notified because that machine will update its cluster-mates. Therefore, only N/2 + 1 propagation messages are sent by the server handling the password registration, saving time. This effect can be further increased by adding more clusters and hierarchical clustering based on regions. Furthermore, we employ threading when updating other clusters in register and delete operations for faster propagation as multiple clusters can have server nodes performing the update at the same time. Figure 3a below depicts this concurrent update propagation to other

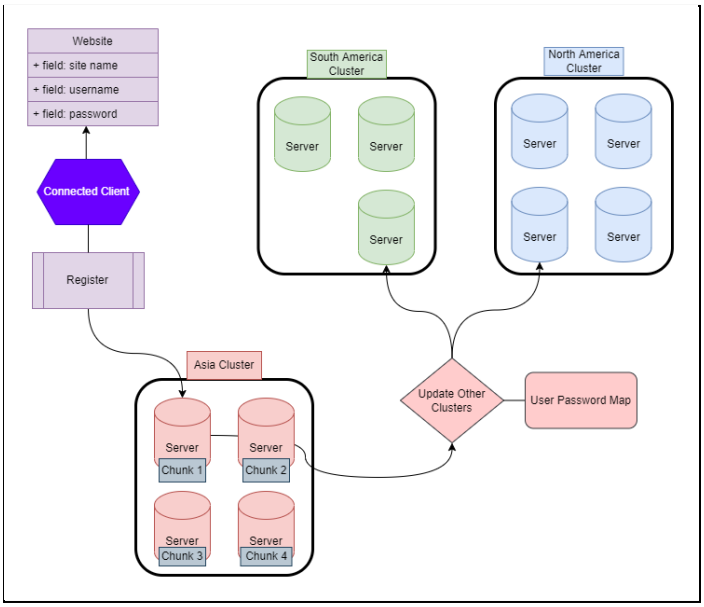


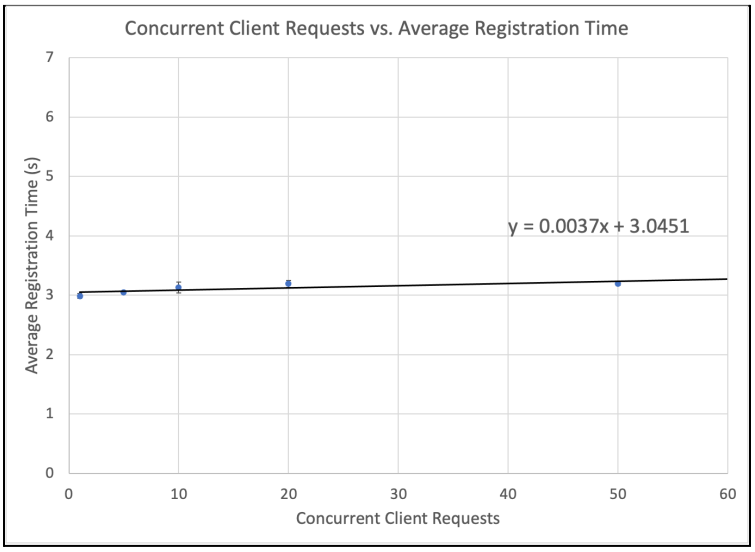
Fig. 3a (Register function): initial server node updating other clusters of a new user account.

clusters. Search operations will function in the same way without the need for multithreading as password chunks for a single user account are typically stored within a single cluster. To elaborate further, users can log on to any server node and request that one of their passwords be retrieved. The initial server node will then check the User Password map to find the cluster and server machines where that client's password chunks are stored and send a request to a relevant node in that cluster to gather the client's full password. Once that password is retrieved, it will be returned to the initial server node which passes that information back to the client. It's important to note that the current system doesn't implement this kind of cluster-based search. Rather, the initial server node reaches out to every node that is storing a chunk of the password. Since clients will most likely connect at the same cluster which stores their passwords, this will essentially result in a cluster-based search. Nonetheless, the current search function needs to be updated so that passwords can always be efficiently retrieved in a cluster-based fashion, regardless of the node that is serving the client.

Evaluation

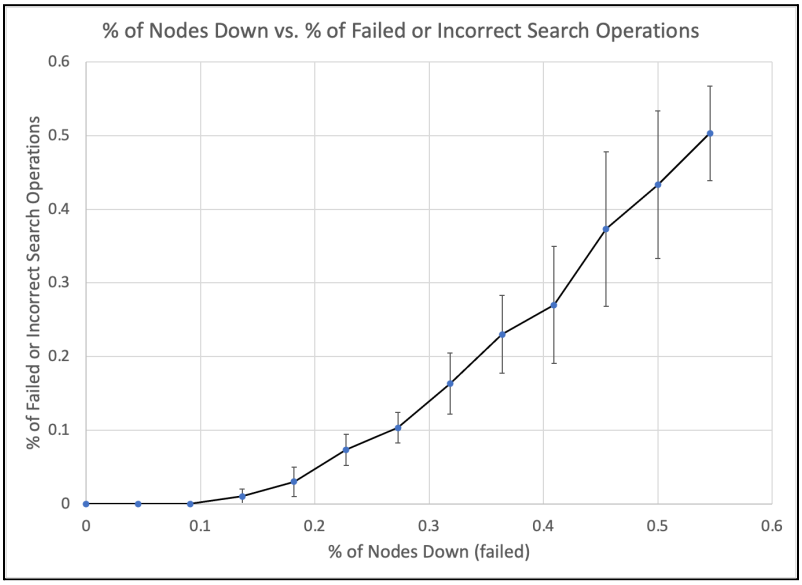
To evaluate our system, we tested its performance under four different conditions: 1) heavy load of clients, 2) Node failures, 3) when GPS is storing large amounts of data, and 4) when passwords are split into many chunks.

For the first three experiments we used 4 as the number of chunks per password. When calculating average registration time, we repeated three register operations and averaged their runtimes.



That process was repeated three times to produce the error bars (a total of nine repetitions per data point).

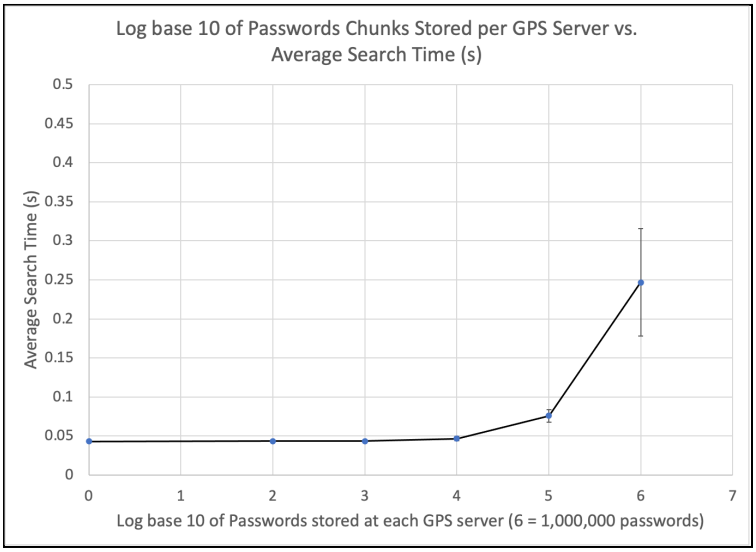
The graph above shows how password registration time varies with concurrent client requests (to one server). Since we use threading, this (almost) horizontal line is expected and ideal. While the error bars show that the trendline might not be sloped at all, we expect that registration will take marginally more time when the server is under heavy load because of the overhead of threading and the machine's limited processing resources. The trendline slope of 0.0037 signifies that adding 270 clients would only cost 1 second per registration (if this trend continues). While we can't say for sure how many concurrent requests GPS can handle, we know that at least 2200 requests (across the 44 machines) will run in nearly the same time as 1 request. This statistic shows that our system is scalable enough to handle a reasonable workload of concurrent requests.



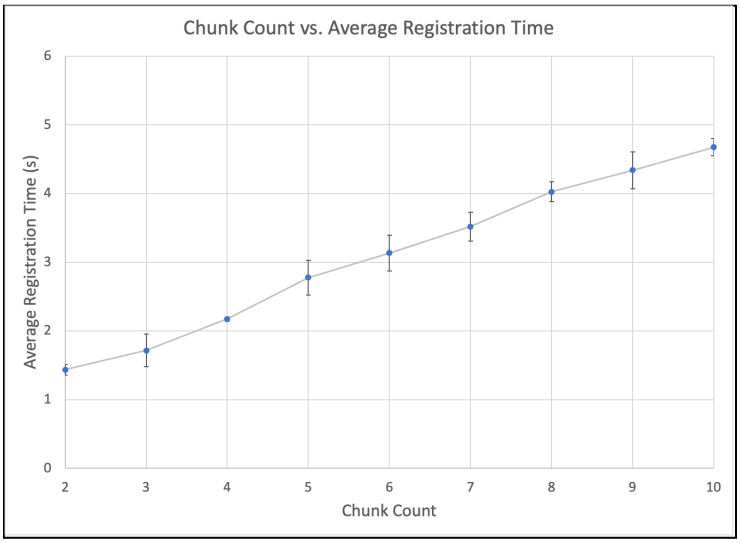
This second graph (shown right) shows how GPS's accuracy diminishes when nodes fail. Despite only replicating data once (every password chunk is stored on two machines), our system can still handle ~20% node failure with almost no data loss. However, as we approach 50% node failure, almost half of the passwords are lost (meaning that both of the machines storing one chunk are down). We have added a working node failure algorithm that replicates the lost data, but we weren't able to test the functionality at scale. In principle, this

addition should mean that node failures will cause search operations to take much longer (while the data is copied over), but never fail or return the wrong answer.

Thirdly, we wanted to know if GPS can handle large amounts of data. We tested our system by manually placing set numbers of password chunks on each server and then running search operations. This graph (below) is on a log scale, and the real scale is from 0 to 1 million



password chunks per machine. We see almost no change in search time until each node stores 100,000 passwords, at which point search time increases by about 70% (but is still less than 0.1 seconds). However, at 1 million chunks per node, the search time increases by 500%. Possible reasons for this spike include Python dictionary implementations, machine memory space, caching, or just the general performance costs of searching through massive amounts of passwords. This data proves that even with all of the information stored in memory, GPS can handle lots of passwords without a big performance cost. If we saved everything to disk, our runtime wouldn't increase by nearly as much, but it would also be much higher when GPS is storing little data, as disk operations are more costly.



Fourthly, we tested the effect of splitting passwords into different numbers of

chunks on Registration time. As expected, this time increases when more chunks are used because GPS needs to make more RPCs and propagate larger updates to each machine. Since our register method returns before inter-cluster propagation and does that part in the background, the latter reason matters less. However, sending extra RPCs to store chunks on different machines does take longer. We think that 3 or 4 chunks is ideal for the average GPS user, while 8 or 10 chunks may be better for users who want to be extra careful and are willing to wait a few more seconds.

Conclusion

Global PassStore (GPS) is a scalable distributed password management system that offers high availability, fault-tolerance, and sufficient performance. GPS clients are able to register, delete, and search for their passwords in an efficient manner from any GPS server node. Our evaluation tests show that GPS is scalable enough to handle high client traffic as with similar request times for increasing concurrent client requests. Our system also scales well in regards to the amount of data being stored as password search times continue to remain low despite the increasing amounts of passwords being stored on each server node. Continuing, GPS employs relatively graceful degradation with little to no data loss even if 10-30% of the servers fail. Furthermore, data loss can be reduced even further by replicating all password chunks more than once; in this implementation we replicate just once. Lastly, we note that the number of chunks that passwords are being split into has the impact of increasing the average register time for a given password. Regardless, we believe that the default chunking value of 4 is sufficient for most clients that seek our system's security benefits of distributed password storage.

Figure 5a can be referenced below for a visual depiction of the fault-tolerance mechanism for node faults.

Next Steps:

Since no distributed system is ever truly complete, let us discuss next steps for GPS. Registration time could be reduced by integrating more dynamic node clusters. These clusters ought to be automatically formed and reformed based on the ping times between nodes: in this way, GPS would be even more optimized for reducing network latency which would reduce the time it takes to register a password. We could also automatically change cluster configuration based on the conditions of the system: if one cluster becomes too small to store passwords, we could combine it with another cluster. This and similar functionality would give the system another degree of flexibility.

In addition to the node addition and removal mechanisms, GPS needs to go above and beyond the remaining security features that are normally expected of other password management systems. We implement simple username-password based authentication, but we fail to account for attacks such as simultaneous registering of a new username/password combination. Continuing, no password manager would be complete without encryption of client data including usernames, websites, and passwords. The way we propose that this be done is through a GPS smart client that encrypts any user data before passing it along to server nodes, and decrypts any encrypted data received from GPS servers before returning that data to users. Server nodes could even encrypt data with each other, with each server having a different public/private keypair with every other server. This may however start to have large storage

overhead as the system grows in size. These and other security flaws in this system would need to be addressed here if GPS seeks to be a full-fledged password manager.

Additionally, more attention must be given to update ordering and related issues. It's possible that simultaneous updates could, in effect, cancel each other out. One possible way to avoid this is a GFS-like system where each cluster has a rotating "update manager" that will keep track of update ordering, and tell the nodes in each cluster the order to apply updates in. Alternatively, a lamport clock-based system could be used to better determine order.

We also hope to employ threading to a greater degree to speed up making multiple RPC's. The desired functionality would be to have a set number of workers, and to have each one making RPC's until all are finished. However, this could not be implemented in the time frame; instead, we employ a mixture of (slow) one-by-one RPC's and (faster) making RPC's in groups of threads, where each group waits for all threads in the previous one to finish.,

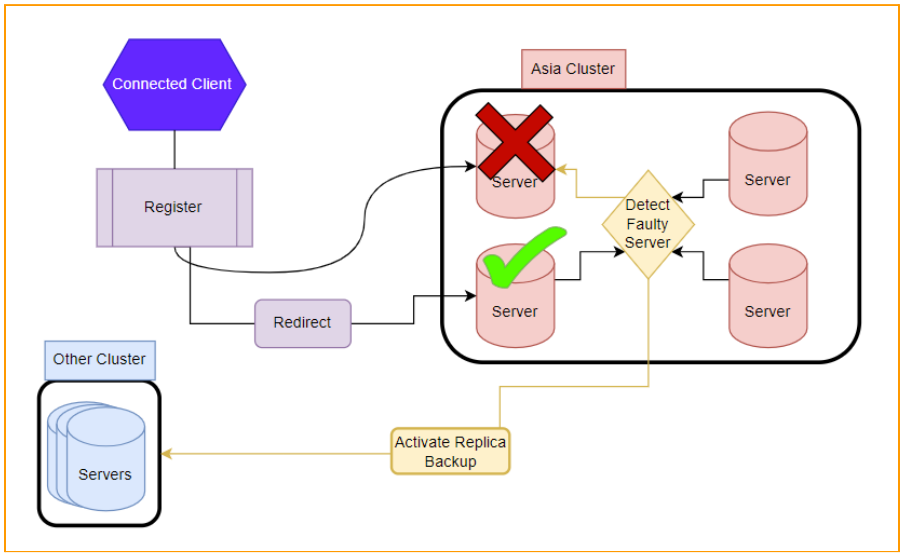


Fig. 5a: GPS faulty server detection and activation of replica backups to recreate the server's local password data.

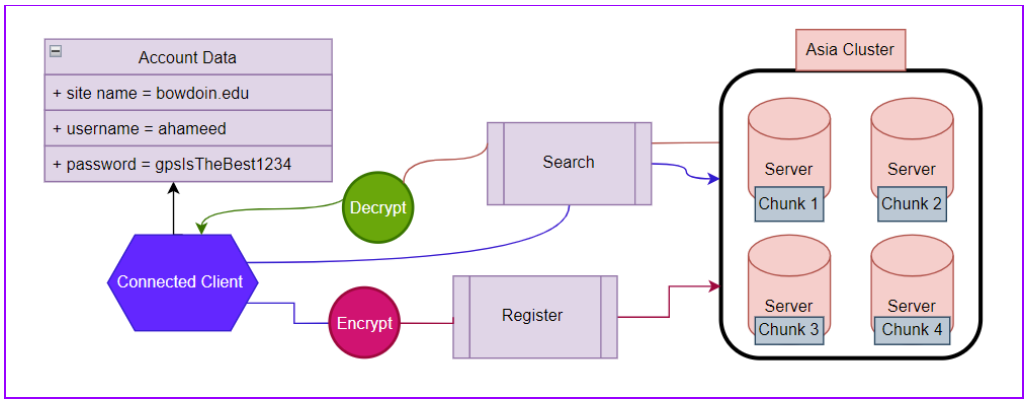


Fig. 5b: GPS smart client encryption and decryption of user account data and passwords (unimplemented).

All in all, GPS provides the foundation for a truly distributed password manager and this foundation can be built upon to create a full software application that is built to scale, across the globe. Although its functionality is not fully complete as a password manager, its design choices make for a promising application that offers unique security features.