Big Data, Small Proof

2/26/2013

1 Entity Notation

Entity	Meaning
AS	Analytic server
WS	Web server
CL	Client who owns the web site associated to analytic service
WB	Web browser of web site users

2 Design

2.1 Architecture Overview

Figure 1 shows the components of the project. Shaded entities are implemented/modified in the project.

Analytic Server stores analytic data from user web browsers and manages SADS to prove the correctness of the analytic data. Upon receiving analytic data from website user's web browser, the analytic server i) stores the received analytic data in the database and ii) updates the SADS. Also, the analytic server handles queries from clients. When the analytic server receives a query from the client, it responds with the result of the query along with the proof derived from SADS.

Web Server hosts the website of the client where analytic Javascript is embedded. The web server also deals with analytic data received from website user's web browser and then it builds the digest based on the received analytic data. The digest is for verifying the proof of the answer of analytic server.

When *Website User* accesses the website using a commodity web browser, and the analytic Javascript is transferred to the web browser along with the website contents. The web browser collects analytic data by running analytic Javascript and sends the collected analytic data to both analytic server and web server through *2-Phase commit* protocol. 2-Phase commit protocol is described in 3.3.

Client is the owner of the website associated with the analytic server. Client sends a query to the analytic server and receives an answer to the query along with the proof.

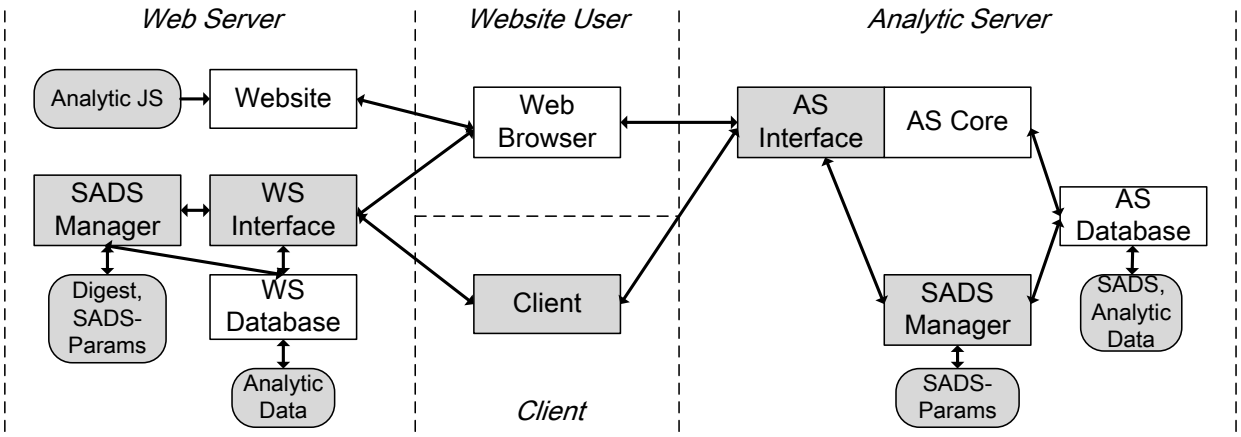


Figure 1: Architecture of the project. Rectangular and rounded boxes represent processes and data, respectively. Shading indicates components added or modified by the project.

Then he verifies the answer and the proof using the digest received from the web server. If any inconsistency between the digest of the web server and SADS of the analytic server is found, the client executes *reconcile* protocol to recover the consistency. Reconcile protocol is described in 3.6.

2.2 Analytic Server

2.2.1 AS Interface

The genuine analytic server such as Google Analytics and Piwik consists of *AS Core* and AS interface. The analytic server provides *AS Interface* for receiving analytic data from web browsers and for handling queries from the client. AS interface is a part of analytic server program but we logically separate it from the analytic server program to integrate SADS into the original analytic server program.

Basically, AS interface receives analytic data from the website users' web browsers and forwards to *AS core* which processes the analytic data. In addition to this genuine function, AS interface forwards whole (or part of) analytic data to *SADS manager* when it is confirmed that the web server also has received the analytic data. AS interface also forwards queries and other messages related to reconciliation process to SADS manager.

2.2.2 AS Core

AS Core is a part of the genuine analytic server. It processes analytic data and queries from the client. In this project, AS core performs its genuine functions and therefore it is not modified.

2.2.3 SADS Manager

SADS manager in analytic server maintains SADS-related parameters such as security parameter k, matrices $\mathbf{L}$ and $\mathbf{R}$ and other necessary parameters while the whole tree structure for SADS and necessary partial digest table are stored in *Database*. It handles analytic data and update SADS stored in the database. It also processes queries received through AS interface and generates corresponding answers and proofs. In addition, it performs reconciliation process when it receives relevant message. Whenever it updates values of leaf nodes of SADS tree, it should check the log of analytic data stored by AS core so that SADS and stored analytic data are synchronized.

2.2.4 AS Database

AS database maintains log of received analytic data and the table for SADS tree. Log of analytic data is managed by AS core and it is not modified in this project.

A table for SADS tree consists of two columns, <*ID, Value*>. Assuming that the size of the universe $\mathcal{U}$ is $M = 2^m$, the size of *ID* is set to $(M+1)$ bits. IDs of leaf nodes are expressed as in the form of $(1||nodeid)$ where *nodeid* is M-bit string for 2^M leaf nodes. Then the ID of parent node of an arbitrary child node can be set to $(1||\lfloor nodeid/2 \rfloor)$. As an example, if we want to build a SADS tree that stores visiting count per IP address, 33 bits are necesary for ID of each node since *nodeid* of a leaf node is 32-bit IP address of a user.

2.3 Web Server

2.3.1 WS Interface

AS interface depicted in 2.2.1 would be reused as *WS interface*. However, WS interface would need partial functionalities of AS core since it needs to process received analytic data and store it into WS database.

2.3.2 SADS Manager

SADS manager in web server also maintains SADS-related parameters, as SADS manager in analytic server does. Upon receiving analytic data, SADS manager updates its digest using SADS-related parameters. When SADS manager receives a request for the digest, it returns its digest to the client. Also it needs to handle messages related to reconciliation process.

2.4 Client

Client sends a query to analytic server and gets an answer and its proof. Also it requests the digest from web server. Then client performs verification process to verify if the answer and the proof received from the analytic server match to the digest received from the web server. If verification fails, the client performs reconciliation process by comparing partial list

of analytic data of analytic server and web server. Reconciliation process is further described in 3.6.

3 Protocol: SADS

When *streaming authenticated data structure (SADS)* is used, analytic server and web servers can update its own digest independently from each other. Therefore, web servers are able to maintain a small digest only, instead of maintaining a whole SADS tree.

The protocol can be used for any case of single or multiple web servers. If multiple web servers participate in the protocol, client is assumed to know IP addresses of all participating web servers. Analytic server is assumed to maintain one SADS per each web server. However, maintaining only one SADS for multiple web servers is also possible, since digests of web servers can be summed up at verification by client due to its homomorphic property.

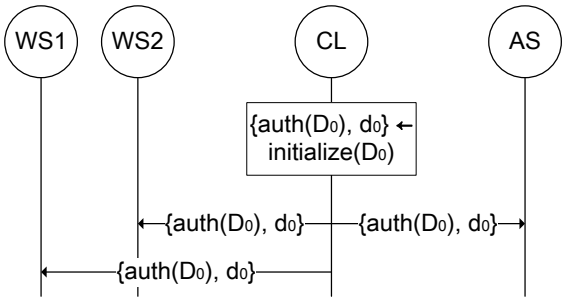
Since analytic data is transmitted to analytic server and web server, two servers should be always synchronized to prevent (or minimize) inconsistency. In this context, *2-phase commit* protocol is proposed for transmission of analytic data. This protocol also makes inconsistencies between analytic server and web servers reconcilable. Once a user's web browser receives the analytic Javascript, it collects analytic data from the user's machine and sends the analytic data with IP address of the web server to analytic server. Analytic data is sent as parameters of HTTP request for a small gif (Google Analytics) or PHP (Piwik) files. Therefore the web browser is able to determine if the transmission of analytic data is successful or not. If the transmission to the analytic server is unsuccessful, a user's web browser terminates the protocol immediately and may restart the protocol. The analytic server defers updating its SADS until it receives confirm message from the user's web browser. If successful, the user's web browser sends the analytic data to the web server. Receiving analytic data from, the web server updates its digest. If transmissions to both web server and analytic server are successful, then the user's web browser sends confirm message to analytic server and it updates SADS.

In this protocol, an inconsistency between the web server and the analytic server can arise only when confirm message to analytic server is not delivered successfully. Therefore the protocol guarantees that web server always have all the analytic data confirmed by analytic server. This provides a basis for reconciliation process described in 3.6.

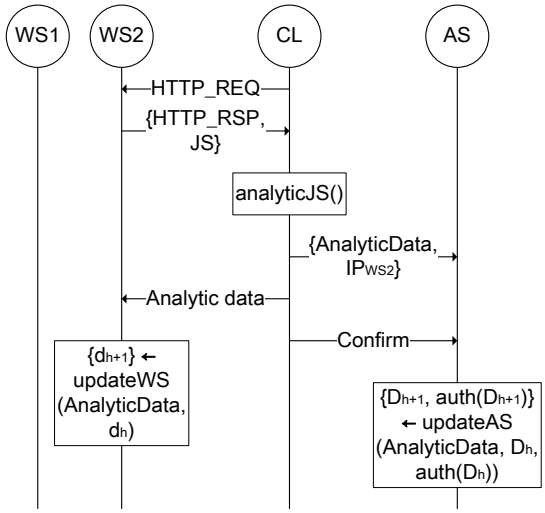
The overall protocol is depicted in Figure 2.

3.1 Initialization

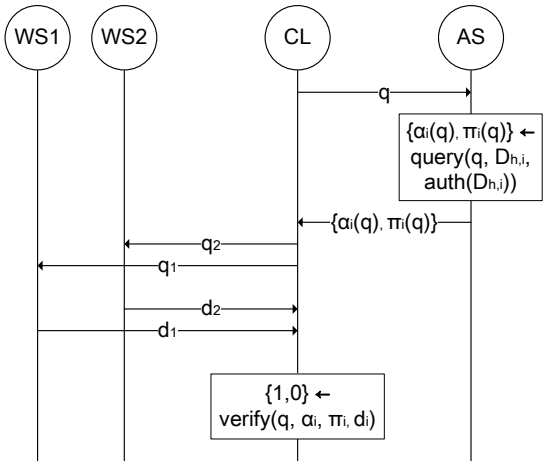
CL: $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{initialize}(D_0)$
CL $\rightarrow$ **AS**: $\{\mathsf{auth}(D_0), d_0\}$
CL $\rightarrow$ **WS**: $\{\mathsf{auth}(D_0), d_0\}$



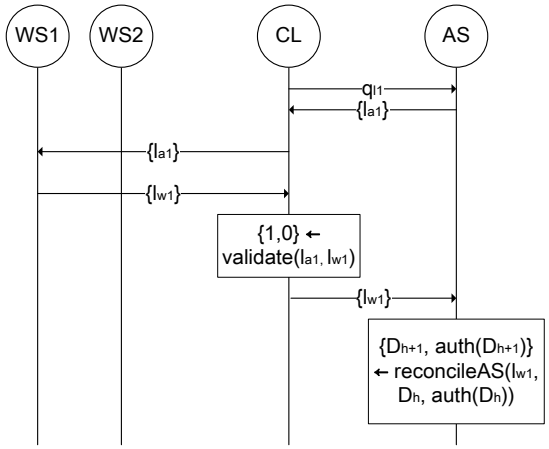
(a) initialize()



(b) 2-Phase commit



(c) query() and verify()



(d) reconcile()

Figure 2: Protocol w/ SADS

Let D_0 be an initial empty dictionary. initialize() outputs SADS auth(D_0) and its digest d_0. Client CL sends {auth(D_0), d_0} to analytic server AS and web servers WS_i ($\forall i$, $i = \{1, ..., N_{ws}\}$ where N_{ws} is the number of web servers hosting the web site).

3.2 Analytic data collection

WB $\rightarrow$ **WS**$_i$: {HTTP_REQ}
WS$_i$ $\rightarrow$ **WB**: {HTTP_RSP, JS}
WB: {AnalyticData} $\leftarrow$ analyticJS()

A user's web browser WB sends HTTP_REQ to WS_i to access the web site. WS_i transmits HTTP_RSP with analytic Javascript JS to WB. Then WB collects AnalyticData by running received analytic Javascript code analyticJS().

3.3 Update: 2-Phase commit

1. 1st Phase
   WB $\rightarrow$ **AS**: {AnalyticData, IP_{WS_i}}
   WB $\rightarrow$ **WS**: {AnalyticData}
   WS: $\{d_{h+1}\}$ $\leftarrow$ updateWS(AnalyticData, d_h)

2. 2nd Phase
   WB $\rightarrow$ **AS**: {Confirm}
   AS: $\{D_{h+1}$, auth(D_{h+1})$\}$ $\leftarrow$ updateAS(AnalyticData, D_h, auth(D_h))

WB sends AnalyticData with IP address of WS_i IP_{WS_i} to AS. If succeed, WB sends AnalyticData to WS. WS performs updateWS() which outputs an updated digest d_{h+1}.
If the 1st phase is performed successfully, WB sends a confirmation message Confirm to AS. Then AS performs updateAS() which outputs an updated dictionary D_{h+1} and SADS auth(D_{h+1}).

3.4 Query

CL $\rightarrow$ **AS**: $\{q\}$
AS: $\{\alpha(q), \Pi(q)\}$ $\leftarrow$ query(q, D_h, auth(D_h))

AS $\rightarrow$ **CL**: $\{\alpha(q), \Pi(q)\}$

CL $\rightarrow$ **WS**: $\{q_d\}$
WS $\rightarrow$ **CL**: $\{d\}$

Let $\alpha(q)$ denote an answer to a query q and $\Pi(q)$ denote a proof of the answer $\alpha(q)$. Query q can be either membership or range query. CL sends a query q to AS. Upon receiving q, AS performs query() and outputs $\{\alpha(q), \Pi(q)\}$. Then AS sends $\{\alpha(q), \Pi(q)\}$ to CL.
On the other hand, CL sends a query q_d to WS to request a digest d. WS sends its digest d to CL.

3.5 Verification

CL: $\{0, 1\} \leftarrow$ verify(q, $\alpha(q)$, $\Pi(q)$, d)

CL performs verify() to check if $\alpha(q)$ and $\Pi(q)$ is authenticated correctly and the digest derived from $\Pi(q)$ matches to d received from the web server. verify() outputs 1 if $\{\alpha(q), \pi(q)\}$ are proved to be authenticated correctly and otherwise, outputs 0.

3.6 Reconciliation

CL $\rightarrow$ **AS**: $\{q_\ell\}$
AS $\rightarrow$ **CL**: $\{\ell_a\}$

CL $\rightarrow$ **WS**: $\{\ell_a\}$
WS $\rightarrow$ **CL**: $\{\ell_w\}$

CL: $\{0,1\} \leftarrow$ validate(ℓ_a, ℓ_w)

CL $\rightarrow$ **AS**: $\{\ell_w\}$
AS: $\{D_{h+1}$, auth(D_{h+1})$\} \leftarrow$ reconcileAS(ℓ_w, D_h, auth(D_h))

Let q_ℓ denote a query requesting the list of AnalyticData ℓ_a that are not paired with Confirm message from AS. If verify() outputs 0, CL sends q_ℓ to AS and receives ℓ_a from AS. Then CL forwards ℓ_a to WS. WS selects from ℓ_a the list of AnalyticData that were actually received. Then WS generates ℓ_w, a sublist of ℓ_a, and sends it back to CL.

CL performs validate() to check if the current inconsistency between AS and WS is recoverable. validate() outputs 1 if ($|\ell_a| - |\ell_w| < Threshold$) and 0 otherwise. If the inconsistency is reconcilable, CL sends ℓ_w to AS, and AS performs reconcileAS() which outputs updated $\{D$, auth(D)$\}$

4 Protocol: Merkle Hash Tree

If *Merkle hash tree* is used as the authenticated data structure, both analytic server and web servers should maintain Merkle hash trees since updating digest without interaction between web servers and analytic server is not possible. We can try to sync two Merkle hash trees to maintain the size of trees as small as possible but it doesn't change the fact that web servers should always maintain its own Merkle hash tree. So using Merkle hash tree in this project seems to be inefficient in terms of space and computation of WS.

As in the protocol with SADS, analytic server maintains one Merkle hash tree per each web server. However, Merkle hash trees of web servers can be merged into a single Merkle hash tree at query time. Therefore, analytic server may maintain only one Merkle hash tree if web servers maintain their own Merkle hash trees.

The overall protocol is depicted in Figure 3.

4.1 Initialization

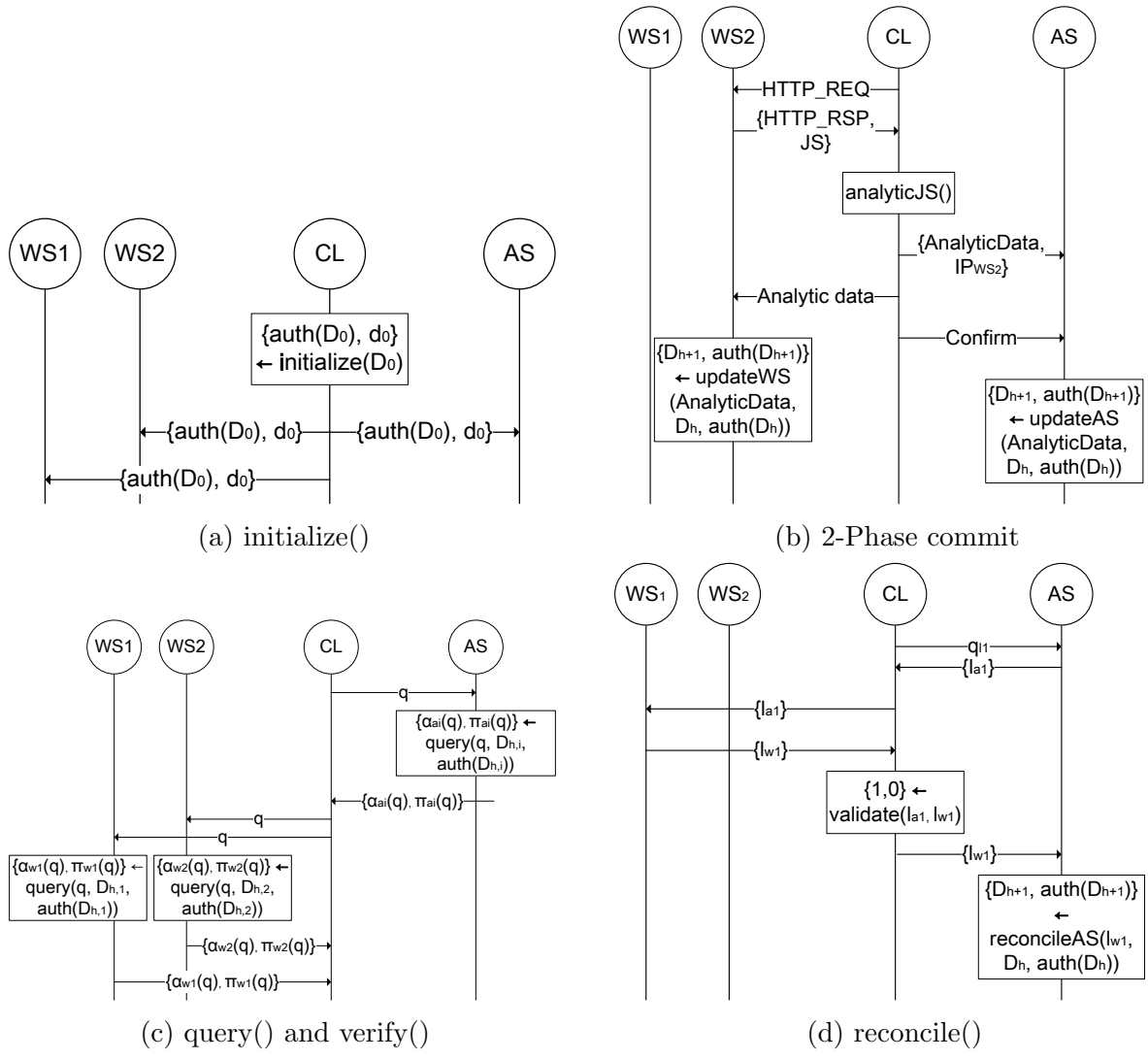
CL: $\{$auth(D_0), $d_0\} \leftarrow$ initialize(D_0)
CL $\rightarrow$ **AS**: $\{$auth(D_0), $d_0\}$
CL $\rightarrow$ **WS**: $\{$auth(D_0), $d_0\}$

Let D_0 be an initial empty dictionary. initialize() outputs Merkle hash tree auth(D_0) and its digest d_0. Client CL sends $\{$auth(D_0), $d_0\}$ to analytic server AS and web servers WS_i ($\forall i$, $i = \{1, ..., N_{ws}\}$ where N_{ws} is the number of web servers hosting the web site).

4.2 Analytic data collection

WB $\rightarrow$ **WS**$_i$: {HTTP_REQ}
WS$_i$ $\leftarrow$ **WB**: {HTTP_RSP, JS}
WB: {AnalyticData} $\leftarrow$ analyticJS()

A user's web browser WB sends HTTP_REQ to WS_i to access the web site. WS_i transmits HTTP_RSP with analytic Javascript JS to WB. Then WB collects AnalyticData by running received analytic Javascript code AnalyticJS().



(a) initialize()

(b) 2-Phase commit

(c) query() and verify()

(d) reconcile()

Figure 3: Protocol w/ Merkle Hash Tree

4.3 Update: 2-Phase commit

1. 1st Phase
   WB $\rightarrow$ **AS**: {AnalyticData, IP_{WS_i}}
   WB $\rightarrow$ **WS**: {AnalyticData}
   WS: $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{updateWS}(\mathsf{AnalyticData}, D_h, \mathsf{auth}(D_h))$
2. 2nd Phase
   WB $\rightarrow$ **AS**: {Confirm}
   AS: $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{updateAS}(\mathsf{AnalyticData}, D_h, \mathsf{auth}(D_h))$

WB sends AnalyticData with IP address of WS_i IP_{WS_i} to AS. If succeed, WB sends AnalyticData to WS. WS performs updateWS() which outputs an updated dictionary D_{h+1} and Merkle hash tree $\mathsf{auth}(D_{h+1})$.
If the 1st phase is done successfully, WB sends a confirmation message Confirm to AS. AS performs updateAS() which outputs an updated dictionary D_{h+1} and Merkle hash tree $\mathsf{auth}(D_{h+1})$.

4.4 Query

CL $\rightarrow$ **AS**: $\{q\}$
AS: $\{\alpha_a(q), \Pi_a(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h))$
AS $\rightarrow$ **CL**: $\{\alpha_a(q), \Pi_a(q)\}$

CL $\rightarrow$ **WS**: $\{q\}$
WS: $\{\alpha_w(q), \Pi_w(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h))$
WS $\rightarrow$ **CL**: $\{\alpha_w(q), \Pi_w(q)\}$

Let $\alpha(q)$ denotes an answer to a query q and $\Pi(q)$ a proof of the answer $\alpha(q)$. CL sends a query q to AS and WS. Upon receiving q, AS and WS perform query() and output $\{\alpha_a(q), \Pi_a(q)\}$ and $\{\alpha_a(q), \Pi_a(q)\}$, respectively. Then AS sends $\{\alpha_a(q), \Pi_a(q)\}$, and WS sends $\{\alpha_w(q), \Pi_w(q)\}$ to CL.

4.5 Verify

CL: $\{0, 1\} \leftarrow \mathsf{verify}(q, \alpha_a(q), \Pi_a(q), \alpha_w(q), \Pi_w(q))$

CL performs verify() to check if $\Pi_a(q)$ and $\Pi_a(q)$ are authenticated correctly and if $\alpha_a(q)$ and $\alpha_w(q)$ are identical.

4.6 Reconcile

CL $\rightarrow$ **AS**: $\{q_\ell\}$
AS $\rightarrow$ **CL**: $\{\ell_a\}$

CL $\rightarrow$ **WS**: $\{\ell_a\}$
WS $\rightarrow$ **CL**: $\{\ell_w\}$

CL: $\{0,1\} \leftarrow \mathsf{validate}(\ell_a, \ell_w)$

CL $\rightarrow$ **AS**: $\{\ell_w\}$
AS: $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{reconcileAS}(\ell_w, D_h, \mathsf{auth}(D_h))$

Let q_ℓ denote a query requesting the list of AnalyticData ℓ_a that are not paired with Confirm message from AS. If verify() outputs 0, CL sends q_ℓ to AS and receives ℓ_a from AS. Then CL forwards ℓ_a to WS. WS selects from ℓ_a the list of AnalyticData that were actually received. Then WS generates ℓ_w, a sublist of ℓ_a, and sends it back to CL.
CL performs validate() to check if the current inconsistency between AS and WS is recoverable. validate() outputs 1 if $(|\ell_a| - |\ell_w| < Threshold)$ and 0 otherwise. If the inconsistency is reconcilable, CL sends ℓ_w to AS, and AS performs reconcileAS() which outputs updated $\{D, \mathsf{auth}(D)\}$