How to deploy a Data Availability Server (DAS)

PUBLIC PREVIEW DOCUMENT This document is currently inpublic preview and may change significantly as feedback is captured from readers like you. Click theRequest an update button at the top of this document orjoin the Arbitrum Discord to share your feedback. AnyTrust chains rely on an external Data Availability Committee (DAC) to store data and provide it on-demand instead of using itsparent chain as the Data Availability (DA) layer. The members of the DAC run a Data Availability Server (DAS) to handle these operations.

In this how-to, you'll learn how to deploy a DAS that exposes:

1. An RPC interface
2. that the sequencer uses to store batches of data on the DAS.
3. An HTTP REST interface
4. that lets the DAS respond to requests for those batches of data.

For more information related to configuring a DAC, refer to theIntroduction .

This how-to assumes that you're familiar with:

- The DAC's role in the AnyTrust protocol. Refer tdnside AnyTrust
- for a refresher.
- Kubernetes
- . The examples in this guide use Kubernetes to containerize your DAS.

How does a DAS work?

A Data Availability Server (DAS) allows storage and retrieval of transaction data batches for an AnyTrust chain. It's the software that the members of the DAC run in order to provide the Data Availability service.

DA servers accept time-limited requests to store data batches from the sequencer of an AnyTrust chain, and return a signed certificate promising to store that data during the established time. They also respond to requests to retrieve the data batches.

Configuration options

When setting up a DAS, there are certain options you can configure to suit your infrastructure needs:

Interfaces available in a DAS

There are two main interfaces that can be enabled in a DAS: anRPC interface to store data in the DAS, intended to be used only by the AnyTrust sequencer; and aREST interface that supports only GET operations and is intended for public use.

DA servers listen on two primary interfaces:

1. ItsRPC interface
2. listens fordas_store
3. RPC messages coming from the sequencer. Messages are signed by the sequencer, and the DAS checks this signature upon receipt.
4. ItsREST interface
5. respond to HTTP GET requests pointed at/get-by-hash/
6. . This uses the hash of the data batch as a unique identifier, and will always return the same data for a given hash.

IPFS is an alternative interface that serves requests for batch retrieval. A DAS can be configured to sync and pin batches to its local IPFS repository, then act as a node in the IPFS peer-to-peer network. The advantage of using IPFS is that the Nitro node will use the batch hashes to find the batch data on the IPFS peer-to-peer network. Depending on the network configuration, that Nitro node may then also act as an IPFS node serving the batch data.

Storage options

A DAS can be configured to use one or more of four storage backends:

- AWS S3
- bucket
- Files on local disk
- Badger
- database on local disk
- IPFS

If more than one option is selected, store requests must succeed to all of them for it to be considered successful, while retrieve requests only require one of them to succeed.

If there are other storage backends you'd like us to support, send us a message onDiscord , or contribute directly to theNitro repository .

Caching

An in-memory cache can be enabled to avoid needing to access underlying storage for retrieve requests.

Requests sent to the REST interface (to retrieve data from the DAS) always return the same data for a given hash, so the result is cacheable. It also contains acache-control header specifying that the object is immutable and to cache it for up to 28 days.

State synchronization

DA servers also have an optional REST aggregator which, when a data batch is not found in cache or storage, requests that batch to other REST servers defined in a list and stores that batch upon receiving it. This is how a DAS that misses storing a batch (the AnyTrust protocol doesn't require all of them to report success in order to post the batch's certificate to the parent chain) can automatically repair gaps in the data it stores, and also how amirror DAS can sync its data. A public list of REST endpoints is published online, which the DAS can be configured to download and use, and additional endpoints can be specified in the configuration.

How to deploy the DAS

Step 0: Prerequisites

Gather the following information:

- The latest Nitro docker image:offchainlabs/nitro-node:v2.3.2-064fa11
- An RPC endpoint for theparent chain
- . It is recommended to use athird-party provider RPC
- orrun your own node
- to prevent being rate limited.
- The SequencerInbox contract address in the parent chain.
- If you wish to configure aREST aggregator for your DAS
- , you'll need the URL where the list of REST endpoints is kept.

Step 1: Generate the BLS keypair

Next, we'll generate a BLS keypair. The private key will be used to sign theData Availability Certificates (DACert) when receiving requests to store data, and the public key will be used to prove that the DACert was signed by the DAS. The BLS private key is sensitive and care must be taken to ensure it is generated and stored in a safe environment.

The BLS keypair must be generated using thedatool keygen utility. Later, it will be passed to the DAS by file or command line.

When running the key generator, we'll specify the--dir parameter with the absolute path to the directory inside the volume to store the keys in.

Here's an example of how to use thedatool keygen utility inside Docker and store the key that will be used by the DAS in the next step.

docker run -v (pwd) /bls_keys:/data/keys --entrypoint datool \ offchainlabs/nitro-node:v2.3.2-064fa11 keygen --dir /data/keys

Step 2: Deploy the DAS

To run the DAS, we'll use thedaserver tool and we'll configure the following parameters:

Parameter Description --data-availability.parent-chain-node-url RPC endpoint of a parent chain node --data-availability.sequencer-inbox-address Address of the SequencerInbox in the parent chain --data-availability.key.key-dir The absolute path to the directory inside the volume to read the BLS keypair ('das_bls.pub' and 'das_bls') from --enable-rpc Enables the HTTP-RPC server listening on --rpc-addr and --rpc-port --rpc-addr HTTP-RPC server listening interface (default "localhost") --rpc-port (Optional) HTTP-RPC server listening port (default 9876) --enable-rest Enables the REST server listening on --rest-addr and --rest-port --rest-addr REST server listening interface (default "localhost") --rest-port (Optional) REST server listening port (default 9877) --log-level Log level: 1 - ERROR, 2 - WARN, 3 - INFO, 4 - DEBUG, 5 - TRACE (default 3) To enable caching, you can use the following parameters:

Parameter Description --data-availability.local-cache.enable Enables local in-memory caching of sequencer batch data --data-availability.local-cache.capacity Maximum number of entries (up to 64KB each) to store in the cache. (default 20000) To enable the REST aggregator, use the following parameters:

Parameter Description --data-availability.rest-aggregator.enable Enables retrieval of sequencer batch data from a list of remote REST endpoints --data-availability.rest-aggregator.online-url-list A URL to a list of URLs of REST DAS endpoints that is checked at startup. This option is additive with the urls option --data-availability.rest-aggregator.urls List of URLs including 'http://' or 'https://' prefixes and port numbers to REST DAS endpoints. This option is additive with the online-url-list option --data-availability.rest-aggregator.sync-to-storage.check-already-exists When using a REST aggregator, checks if the data already exists in this DAS's storage. Must be disabled for fast sync with an IPFS backend (default true) --data-availability.rest-aggregator.sync-to-storage.eager When using a REST aggregator, eagerly syncs batch data to this DAS's storage from the REST endpoints, using the parent chain as the index of batch data hashes; otherwise only syncs lazily --data-availability.rest-aggregator.sync-to-storage.eager-lower-bound-block When using a REST aggregator that's eagerly syncing, starts indexing forward from this block from the parent chain. Only used if there is no sync state. --data-availability.rest-aggregator.sync-to-storage.retention-period When using a REST aggregator, period to retain the synced data (defaults to forever) --data-availability.rest-aggregator.sync-to-storage.state-dir When using a REST aggregator, directory to store the sync state in, i.e. the block number currently synced up to, so that it doesn't sync from scratch each time Finally, for the storage backends you wish to configure, use the following parameters. Toggle between the different options to see all available parameters.

- AWS S3 bucket
- Local Badger database
- Local files
- IPFS

Parameter Description --data-availability.s3-storage.enable Enables storage/retrieval of sequencer batch data from an AWS S3 bucket --data-availability.s3-storage.access-key S3 access key --data-availability.s3-storage.bucket S3 bucket --data-availability.s3-storage.region S3 region --data-availability.s3-storage.secret-key S3 secret key --data-availability.s3-storage.object-prefix Prefix to add to S3 objects --data-availability.s3-storage.discard-after-timeout Whether to discard data after its expiry timeout (setting it to false, activates the “archive” mode) Parameter Description --data-availability.local-db-storage.enable Enables storage/retrieval of sequencer batch data from a database on the local filesystem --data-availability.local-db-storage.data-dir Absolute path of the directory inside the volume in which to store the database (it must exist) --data-availability.local-db-storage.discard-after-timeout Whether to discard data after its expiry timeout (setting it to false, activates the “archive” mode) Parameter Description --data-availability.local-file-storage.enable Enables storage/retrieval of sequencer batch data from a directory of files, one per batch --data-availability.local-file-storage.data-dir Absolute path of the directory inside the volume in which to store the data (it must exist) Parameter Description --data-availability.ipfs-storage.enable Enables storage/retrieval of sequencer batch data from IPFS --data-availability.ipfs-storage.profiles Comma separated list of IPFS profiles to use --data-availability.ipfs-storage.read-timeout Timeout for IPFS reads, since by default it will wait forever. Treat timeout as not found (default 1m0s) Here's an exampledaserver command for a DAS that:

- Enables both interfaces: RPC and REST
- Enables local cache
- Enables aREST aggregator
- Enables AWS S3 bucket storage
- Enables local Badger database storage

daserver --data-availability.parent-chain-node-url "" --data-availability.sequencer-inbox-address ''

'' --data-availability.key.key-dir /home/user/data/keys --enable-rpc --rpc-addr '0.0.0.0' --log-level 3 --enable-rest --rest-addr '0.0.0.0' --data-availability.local-cache.enable --data-availability.rest-aggregator.enable --data-availability.rest-aggregator.online-url-list "" --data-availability.s3-storage.enable --data-availability.s3-storage.access-key "' --data-availability.s3-storage.bucket "" --data-availability.s3-storage.region "' --data-availability.s3-storage.secret-key "" --data-availability.s3-storage.object-prefix "/" --data-availability.local-db-storage.enable --data-availability.local-db-storage.data-dir /home/user/data/badgerdb And here's an example of how to use a k8s deployment to run that command:

apiVersion : apps/v1 kind : Deployment metadata : name : das - server spec : replicas :

1 selector : matchLabels : app : das - server strategy : rollingUpdate : maxSurge :

0 maxUnavailable : 50% type : RollingUpdate template : metadata : labels : app : das - server spec : containers : -

command : - bash -

- c -

| mkdir -p /home/user/data/badgerdb /usr/local/bin/daserver --data-availability.parent-chain-node-url "' --data-availability.sequencer-inbox-address ''

" --data-availability.key.key-dir /home/user/data/keys --enable-rpc --rpc-addr '0.0.0.0' --log-level 3 --enable-rest --rest-addr '0.0.0.0' --data-availability.local-cache.enable --data-availability.rest-aggregator.enable --data-availability.rest-aggregator.online-url-list "" --data-availability.s3-storage.enable --data-availability.s3-storage.access-key "' --data-availability.s3-storage.bucket "" --data-availability.s3-storage.region '" --data-availability.s3-storage.secret-key "" --data-availability.s3-storage.object-prefix "/" --data-availability.s3-storage.discard-after-timeout false --data-availability.local-db-storage.enable --data-availability.local-db-storage.data-dir /home/user/data/badgerdb --data-availability.local-db-storage.discard-after-timeout false image: offchainlabs/nitro-node:v2.3.2-064fa11 imagePullPolicy: Always resources: limits: cpu: "4" memory: 10Gi requests: cpu: "4" memory: 10Gi ports: - containerPort: 9876 hostPort: 9876 protocol: TCP - containerPort: 9877 hostPort: 9877 protocol: TCP volumeMounts: - mountPath: /home/user/data/ name: data readinessProbe: failureThreshold: 3 httpGet: path: /health/ port: 9877 scheme: HTTP initialDelaySeconds: 5 periodSeconds: 5 successThreshold: 1 timeoutSeconds: 1 volumes : -

name : data persistentVolumeClaim : claimName : das - server

Archive DA servers

Archive DA servers are servers that don't discard any data after expiring. Each DAC should have at the very least one archive DAS to ensure all historical data is available.

To activate the "archive mode" in your DAS, set the parameterdiscard-after-timeout tofalse in your storage method. For example:

--data-availability.s3-storage.discard-after-timeout

false --data-availability.local-db-storage.discard-after-timeout = false Note thatlocal-file-storage andipfs-storage don't discard data after expiring, so the optiondiscard-after-timeout is not available.

Archive servers should make use of the--data-availability.rest-aggregator.sync-to-storage options described above to pull in any data that they don't have.

Helm charts

A helm chart is available atArtifactHUB . It supports running a DAS by providing the BLS key and the parameters for your server. Find more information in theOCL community Helm charts repository .

Testing the DAS

Once the DAS is running, we can test if everything is working correctly using the following methods.

Test 1: RPC health check

The RPC interface enabled in the DAS has a health check for the underlying storage that can be invoked by using the RPC methoddas_healthCheck that returns200 if the DAS is active.

Example:

curl -X POST \ -H 'Content-Type: application/json'

\ -d '{"jsonrpc":"2.0","id":0,"method":"das_healthCheck","params":[]}'

\ < YOUR RPC ENDPOINT

Test 2: Store and retrieve data

The RPC interface of the DAS validates that requests to store data are signed by the sequencer's ECDSA key, identified via a call to theSequencerInbox contract on the parent chain. It can also be configured to accept store requests signed with another ECDSA key of your choosing. This could be useful for running load tests, canaries, or troubleshooting your own infrastructure.

Using this facility, a load test could be constructed by writing a script to store arbitrary amounts of data at an arbitrary rate; a

canary could be constructed to store and retrieve data on some interval. We show here a short guide on how to do that.

Step 1: Generate an ECDSA keypair

First we'll generate an ECDSA keypair withdatool keygen . Create a folder inside/some/local/dir to store the ECDSA keypair, for example/some/local/dir/keys . Then rundatool keygen :

datool keygen --dir /some/local/dir/keys --ecdsa You can also use thedocker run command as follows:

docker run --rm -it -v /some/local/dir:/home/user/data --entrypoint datool offchainlabs/nitro-node:v2.3.2-064fa11 keygen --dir /home/user/data/keys --ecdsa

Step 2: Change the DAS configuration and restart the server

Add the following configuration parameter todaserver :

--data-availability.extra-signature-checking-public-key /some/local/dir/keys/ecdsa.pub OR

--data-availability.extra-signature-checking-public-key "0x" And then restart it.

Step 3: Store data signed with the ECDSA private key

Now you can use thedatool utility to send store requests signed with the ECDSA private key:

datool client rpc store --url http://localhost:9876 --message "Hello world" --signing-key /some/local/dir/keys/ecdsa OR

datool client rpc store --url http://localhost:9876 --message "Hello world" --signing-key "0x" You can also use thedocker run command:

docker run --rm -it -v /some/local/dir:/home/user/data --network = "host" --entrypoint datool offchainlabs/nitro-node:v2.3.2-064fa11 client rpc store --url http://localhost:9876 --message "Hello world" --signing-key /home/user/data/keys/ecdsa The above command will output theHex Encoded Data Hash which can then be used to retrieve the data in the next step.

Step 4: Retrieve the stored data

Use again thedatool to retrieve the stored data. Notice that to perform this step you must have the REST interface enabled in the DAS:

datool client rest getbyhash --url http://localhost:9877 --data-hash 0xDataHash You can also use thedocker run command:

docker run --rm -it --network = "host" --entrypoint datool offchainlabs/nitro-node:v2.3.2-064fa11 client rest getbyhash --url http://localhost:9877 --data-hash 0xDataHash If we set0xDataHash to0x052cca0e379137c975c966bcc69ac8237ac38dc1fcf21ac9a6524c87a2aab423 (from the previous step), then the result should be:Message: Hello world

The retention period defaults to 24 hours, but can be configured when callingdatool client rpc store with the parameter--das-retention-period and the number of milliseconds for the retention period.

Test 3: REST health check

The REST interface has a health check on the path/health which will return200 if the underlying storage is working, otherwise503 .

Example:

curl -I < YOUR REST ENDPOINT

/health

Running a mirror DAS

To avoid exposing the REST interface of your main DAS to the public in order to prevent spamming attacks (as explained in[Security considerations](#)), you can choose to run a mirror DAS to complement your setup. The mirror DAS will handle all

public REST requests, while reading information from the main DAS via its (now private) REST interface.

In general, mirror DA servers serve two main purposes:

1. *Prevent the main DAS from having to serve requests for data, allowing it to focus only on storing the data received.*
2. *Provide resiliency to the network in the case of a DAS going down.*

Find information about how to setup a mirror DAS in How to deploy a mirror DAS .

Security considerations

Keep in mind the following information when running the DAS.

A DAS should strive not to miss any batch of information sent by the sequencer. Although it can use a REST aggregator to fetch missing information from other DA servers, it should aim to synchronize all received information directly. To facilitate this, avoid placing any load balancing layer before the DAS, enabling it to handle all incoming traffic.

Taking that into account, there's a risk of Denial of Service attacks on those servers if the endpoint for the RPC interface is publicly known. To mitigate this risk, ensure the RPC endpoint's URL is not easily discoverable. It should be known only to the sequencer. Share this information with the chain owner through a private channel to maintain security.

Finally, as explained in the previous section, if you're also running a mirror DAS, there's no need to publicly expose the REST interface of your main DAS. Your mirrors can synchronize over your private network using the REST interface from your main DAS and other public mirrors.

Other considerations

- *When usingnginx*
- *in the networking stack, a DAS might fail receiving batches that are over a certain size. If this happens, the DAS won't be able to sign any more certificates and the batch poster will receive an error413 Request Entity Too Large*
- *. To prevent this behavior, the parameterclient_max_body_size*
- *from nginx configuration should be configured with a higher value than the default 1M. It's recommended to set it to at least 50M.*

What to do next?

Once the DAS is deployed and tested, you'll have to communicate the following information to the chain owner, so they can update the chain parameters and configure the sequencer:

- *Public key*
- *The https URL for the RPC endpoint which includes some random string (e.g. das.your-chain.io/rpc/randomstring123), communicated through a secure channel*
- *The https URL for the REST endpoint (e.g. das.your-chain.io/rest)*

Optional parameters

Besides the parameters described in this guide, there are some more options that can be useful when running the DAS. For a comprehensive list of configuration parameters, you can rundaserver --help .

Parameter Description --conf.dump Prints out the current configuration --conf.file Absolute path to the configuration file inside the volume to use instead of specifying all parameters in the command

Metrics

The DAS comes with the option of producing Prometheus metrics. This option can be activated by using the following parameters:

Parameter Description --metrics Enables the metrics server --metrics-server.addr Metrics server address (default "127.0.0.1") --metrics-server.port Metrics server port (default 6070) --metrics-server.update-interval Metrics server update interval (default 3s) When metrics are enabled, several useful metrics are available at the configured port, at pathdebug/metrics ordebug/metrics/prometheus .

RPC metrics

Metric Description arb_das_rpc_store_requests Count of RPC Store calls arb_das_rpc_store_success Successful RPC Store calls arb_das_rpc_store_failure Failed RPC Store calls arb_das_rpc_store_bytes Bytes retrieved with RPC Store calls arb_das_rpc_store_duration (p50, p75, p95, p99, p999, p9999) Duration of RPC Store calls (ns)

REST metrics

Metric Description arb_das_rest_getbyhash_requests Count of REST GetByHash calls arb_das_rest_getbyhash_success Successful REST GetByHash calls arb_das_rest_getbyhash_failure Failed REST GetByHash calls arb_das_rest_getbyhash_bytes Bytes retrieved with REST GetByHash calls arb_das_rest_getbyhash_duration (p50, p75, p95, p99, p999, p9999) Duration of REST GetByHash calls (ns) Edit this page Last updatedonMar 26, 2024 Previous How to configure a Data Availability Committee: Introduction Next How to deploy a mirror Data Availability Server (DAS)