Zarr v3 design update

Alistair Miles (@alimanfoo)

19 June 2019

Current status

- Development is ongoing via the core-protocol-v3.0-dev branch in the zarr-specs repo, follow PR#16 for current status.
- Rendered docs from this branch can be viewed on RTFD.
- This is still a straw man, everything is up for discussion.

Design principles

1. Hackable
2. Parallel
3. Distributed

Design principles - hackable

- Easy to implement.
- Easy to extend with new functionality.
- Easy to inspect and manipulate metadata and data with generic tools.

Design principles - parallel

- Think "what happens if two workers do X at the same time"?
- Avoid race conditions.

Design principles - distributed

- Accommodate eventual consistency.

Modular spec architecture

- Core protocol spec
- Protocol extension specs
- Codec specs
- Storage specs

Core protocol spec

- Minimal, easy as possible to do full implementation in any language.
- Aiming for intersection of N5 and Zarr v2 features.
- Defines a variety of **extension points** so can also serve as a foundation for growth and experimentation.

Protocol extension specs

- Each protocol extension gets its own spec.
- Currently the core-protocol-v3.0-dev branch also includes some protocol extension specs, to illustrate the concept.
  - E.g., Datetime data types spec
- Ultimately these extension specs should get split out into separate branches, so we decouple them from the core protocol spec.
- N.B., protocol extensions could also **modify** the core protocol (more on that later).

Codec specs

- Each codec intended for use as a compressor or filter gets it's own spec.
- @@TODO create an example to illustrate the concept.

Storage specs

- Each concrete storage system (e.g., file system, cloud object storage, Zip file, LMDB, ...) gets it's own spec.
- @@TODO create an example to illustrate the concept.

Spec development process - current

- Currently @alimanfoo is acting as editor for the v3.0 core protocol spec.
  - Feedback/comments/ideas/contributions welcome from anyone at any time (PR#16 is probably the best place for comments).
  - Still in a conceptualisation phase, no need for formal decision process as yet.

Spec development process - future

- Ultimately I think we'll need to define a community process for spec development, so that:
  - It's clear how others can contribute.
  - It's clear how decisions get made.
  - It's clear what stage of maturity each spec is at.
- Don't have a solution for that yet, may need advice/help on best approach.

Spec development process - freedoms

- Hopefully the zarr-specs repo can serve as a focus for community spec development.
  - ...and the zarr-specs RTFD site can serve as a discovery point for specs.
- However, don't want to force all spec development down the same route, or force all specs to be published in the same place.
  - This is one reason why currently the core protocol spec makes use of **URIs** in metadata to refer to protocol extensions and codecs - allow freedom for anyone to publish their own spec.

Core protocol - concepts and terminology

Hierachy. Group. Array. Name. Path. Dimension. Shape. Element. Data type. Chunk. Grid. Memory layout. Compressor. Codec. Metadata document. Store.

- Are we comfortable with this terminology and how it is defined?
- Any important missing terms/concepts?

Core protocol - node names

- Each node (array or group) in a hierarchy has a name.
- Node names are used to form node paths.
  - E.g., "/foo/bar" is a path identifying a node named "bar" whose parent is named "foo" whose parent is the root node.

Node names - restrictions

- Node paths are used by users to access nodes and explore/navigate a hierarchy.
- N.B., node paths are also used to form storage keys (see later).
- To try and ensure compatibility with a variety of storage systems, the core protocol currently states fairly heavy restrictions on node names.
  - Includes restriction to ASCII alpha-numeric characters, "-", "_" and ".".

Node names - restrictions

- Are we comfortable with the current restrictions?
- Should we be aiming to support Unicode? Or is that a bridge too far for now?
  - Not sure what full implications would be, but supporting Unicode could make storage specs and implementations harder to develop.

Node names - case (in)sensitivity

- N.B., some file systems are case sensitive, some are not.
- This can (and has) led to bugs with the zarr v2 protocol when used with file system storage on different operating systems.
- Hard to resolve without annoying users (e.g., force all names to lower case) or complicating implementation (e.g., check for case-insensitive name clashes).
- I'm not happy with the spec's current approach to this.

Core protocol - arrays

- Arrays have dimensions, shape and data type.
- Shape (length of dimensions) is finite.
  - But protocol extension could modify this to define behaviour for "open-ended" (i.e., infinite) dimensions.

Core protocol - data types

- Boolean (single byte)
- Integer (signed or unsigned; 1, 2, 4, 8 bytes; little- or big-endian)
- Float (2, 4, 8 bytes; little- or big-endian)
- Any other data type can be defined via a protocol extension
  - E.g., datetime data types

Data types - identifiers

- Each data type needs an identifier for use in metadata documents.
- E.g., "bool", "i1", "<i4", ">u8", "<f2", etc.

Extension data types - fallback

- Any extension data type can define a **fallback** which is a core data type with the same item size.
- Allow for graceful degradation of functionality.
- Should this be called "fallback"? Maybe "base type"?

Core protocol - chunk grids

- A chunk grid defines a set of chunks which contain the elements of an array.
- The chunks of a grid form a tessellation of the array space, which is a space defined by the dimensionality and shape of the array.
  - => Every element of the array is a member of one chunk, and there are no gaps or overlaps between chunks.

Grid types

- In general there are several different possible types of grid.
- The core protocol defines regular grids.
- Other grid types could be defined via protocol extensions, e.g., non-uniform (rectilinear) grid.
- Any grid type must define:
  - How the array space is divided into chunks.
  - A unique identifier for each chunk in the grid (used to form storage keys, see later).

Regular grids

- A grid type where each chunk is a (hyper)rectangle of the same shape.
- I.e., essentially the grid type used in HDF5, Zarr v2 and N5, although different behaviours for edge chunks, see below.
- Each chunk has a grid index, which is a tuple of grid coordinates.
    - E.g., grid index (0, 3, 7) means first chunk along first dimension, fourth chunk along second dimension, 8th chunk along third dimension.

Regular grids - chunk identifiers

- Chunk identifier is formed from grid index.
  - E.g., chunk at grid index (0, 3, 7) has identifier "0.3.7".
- Default separator is "." but can be changed (e.g., to "/" as in N5) in array metadata (see later).

Regular grids - edge chunks

- All chunks have the same shape.
- If the length of any array dimension is not perfectly divisible by the chunk length along the same dimension, the grid will overhang the edge of the array space.
- Spec currently doesn't say any more about how to handle edge chunks, maybe it should?
  - E.g., suggest using array fill value to fill contents of edge chunks beyond the array space.
- Other approaches (e.g., truncated edge chunks) could be defined as a different grid type via a protocol extension.

Regular grids - resizing arrays

- Regular grid supports growing and shrinking an array along any dimension.
  - Growing only requires change to array metadata (update array shape), no chunk data needs to be added or modified.
  - Shrinking requires change to array metadata (update array shape) plus delete any chunks now completely outside the array space.
- Regular grid does not support growing an array in "negative" direction, i.e., prepending.
  - But could define a grid type that does support this via a protocol extension.

Core protocol - chunks

- What is a chunk? Logically, it's an N-dimensional typed array, containing data elements from a region of a Zarr array.
- Chunk shape, data type and memory layout are all defined in the metadata for the Zarr array in which the chunk belongs.
  - I.e., chunks don't need their own metadata (at least when using a regular grid).

Core protocol - chunks

- When reading/writing chunks, expect implementations will use some appropriate class for managing typed memory blocks.
  - E.g., in Python could use NumPy, XND or Arrow (for 1D chunks).
  - E.g., in C could use libxnd.
  - E.g., in C++ could use xtensor.

Chunks - memory layouts

- Memory layout (along with data type) defines binary representation for a chunk.
- Core protocol defines two memory layouts for chunks.
  - C contiguous (row-major).
  - F contiguous (column-major).
- Protocol extensions could define other memory layouts.

Chunks - encoding

- Optionally, an array can be configured with a compressor.
- A compressor is a codec which can be used to encode and decode chunks during storage and retrieval.
- Support for filters (sequence of zero or more codecs applied prior to compressor during encoding) has been moved following discussion to a protocol extension.
- N.B., codecs may be lossy, i.e., decode(encode(x)) round trip doesn't have to be perfect, but must preserve memory layout and data type.

Codecs

- Currently, the core protocol spec defines a codec as a pair of algorithms (encode and decode) which each operate on a sequence of bytes.
- This should probably be modified so that codecs operate on typed memory blocks, i.e., have access to information about data type, item size, chunk shape.

How to define a codec?

- Publish a spec which:
  - Defines the encode and decode algorithms (or cites some existing documents that define them).
  - Defines any configuration parameters (e.g., compression level).
  - States the codec identifier, which must be a URI that dereferences to the codec spec.
    - The codec identifier is used in array metadata, see later.
- @@TODO example codec spec

Core protocol - metadata

- Each Zarr hierarchy is defined via metadata documents.
  - One bootstrap metadata document.
  - Multiple array metadata documents.
  - Multiple group metadata documents.
- Metadata documents are defined using the JSON type system (objects, arrays, strings, numbers).
- Metadata documents are serialised (encoded) for storage.
  - Default encoding is JSON, but protocol extensions can define other encodings for group and array metadata, see below.

Array metadata - example

```
{
    "shape": [10000, 1000],
    "data_type": "<f8",
    "chunk_grid": {
        "type": "regular",
        "chunk_shape": [1000, 100]
    },
    "chunk_memory_layout": "C",
    "compressor": {
        "codec": "http://purl.org/zarr/spec/codec/gzip",
        "configuration": {
            "level": 1
        }
    },
    "fill_value": "NaN",
    "extensions": [],
    "attributes": {
        "foo": 42,
        "bar": "apples",
        "baz": [1, 2, 3, 4]
    }
}
```

Array metadata - example using extension data type

```
{
    ...
    "data_type": {
        "extension": "http://purl.org/zarr/spec/protocol/extensions/
        "type": "<M8[ns]",
        "fallback": "<i8"
    },
    ...
}
```

Array metadata - attributes

- N.B., user attributes are contained within the array metadata document.
- No constraint on content, up to user/application.

Array metadata - attributes

Could be a flat set of name/value pairs, e.g.:

```
{
    ...
    "attributes": {
        "foo": 42,
        "bar": "apples",
        "baz": [1, 2, 3, 4]
    }
}
```

Array metadata - attributes

Could be deeply nested structure, e.g.:

```
{
    ...
    "attributes": {
        "@id" : "arc:arc0",
        "@type" : [ "ome:Arc", "ome:ManufacturerSpec" ],
        "identifier" : "LightSource:1",
        "ome:arcType" : {
            "@id" : "arcType:Xe"
        },
        ...
    }
}
```

(This example probably needs to be fixed to properly use JSON-LD, but hopefully the concept is clear.)

Group metadata - example

Currently nothing but extensions and attributes:

```
{
    "extensions": [],
    "attributes": {
        "spam": "ham",
        "eggs": 42,
    }
}
```

Bootstrap metadata

- Bootstrap metadata is a new concept, not present in either Zarr v2 or N5. Why add it?
- Provides metadata about metadata, e.g., what core protocol version is being used, what metadata encoding is being used.
- Allows for extensions to be declared that modify metadata and/or data layout or encoding or other protocol changes that apply globally to the entire hierarchy.
    - E.g., a protocol extension for consolidated metadata could declare itself here, allowing an implementation to discover that consolidated metadata is available.
    - E.g., a protocol extension that uses something other than JSON for metadata encoding could declare itself here.

Bootstrap metadata - further rationale

- Allows hierarchies to be self-describing (i.e., don't need out-of-band information to interpret).
- Allows implementations to provide appropriate error messages when unsupported protocol extensions/modifications are being used.

Bootstrap metadata - example

```
{
    "zarr_format": "http://purl.org/zarr/spec/protocol/core/3.0",
    "metadata_encoding": "application/json",
    "extensions": [
        {
            "extension": "http://example.org/zarr/extension/foo",
            "must_understand": false,
            "configuration": {
                "foo": "bar"
            }
        }
    ]
}
```

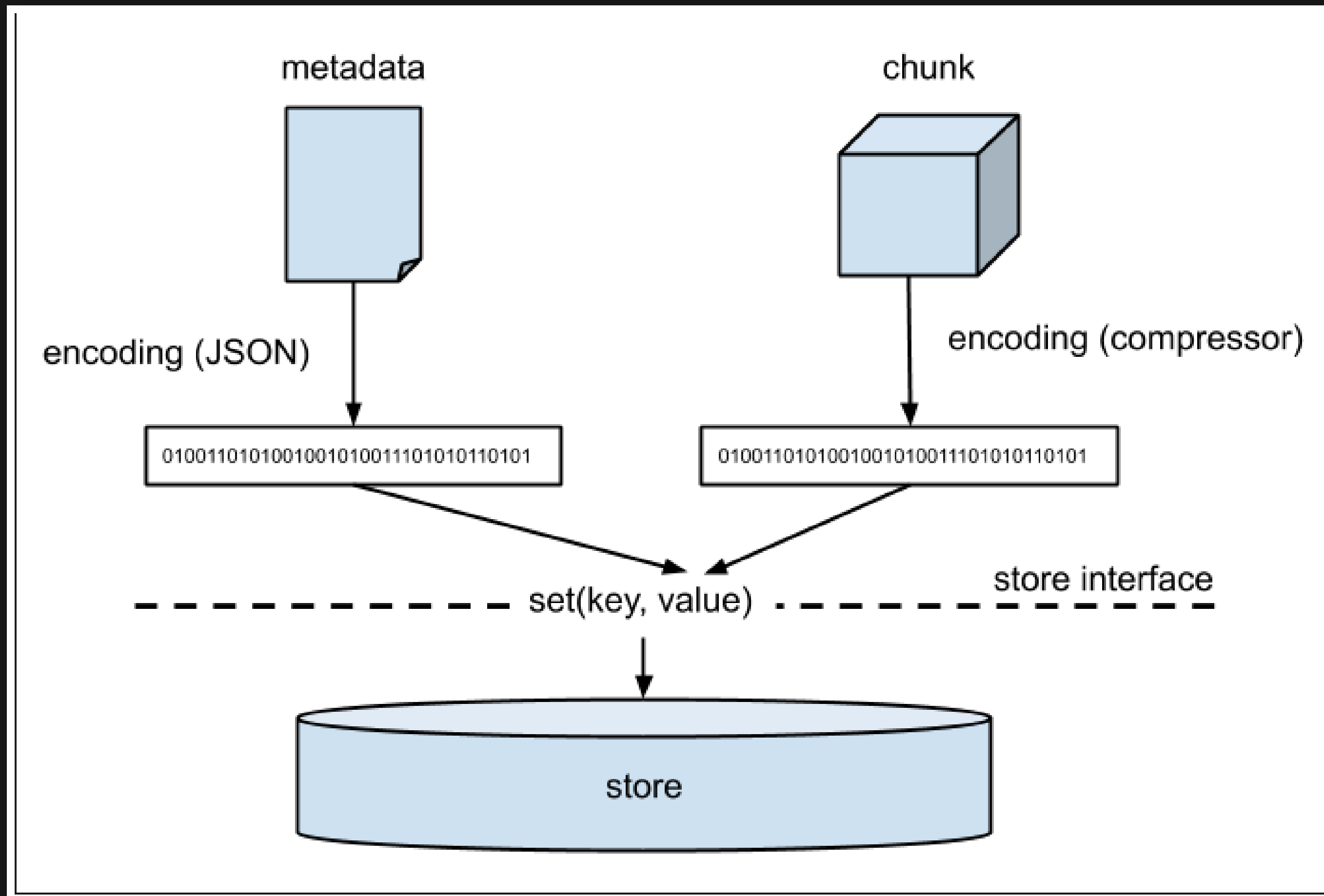
Core protocol - stores

- All stores encapsulated by a simple store interface.
- The store interface comprises a set of operations involving keys and values.
  - A key is an ASCII string (with some restrictions).
  - A value is a sequence of bytes.
- Assume the store holds (key, value) pairs, with only one value for any given key.
  - I.e., a store is a mapping from keys to values.

Store interface - capabilities

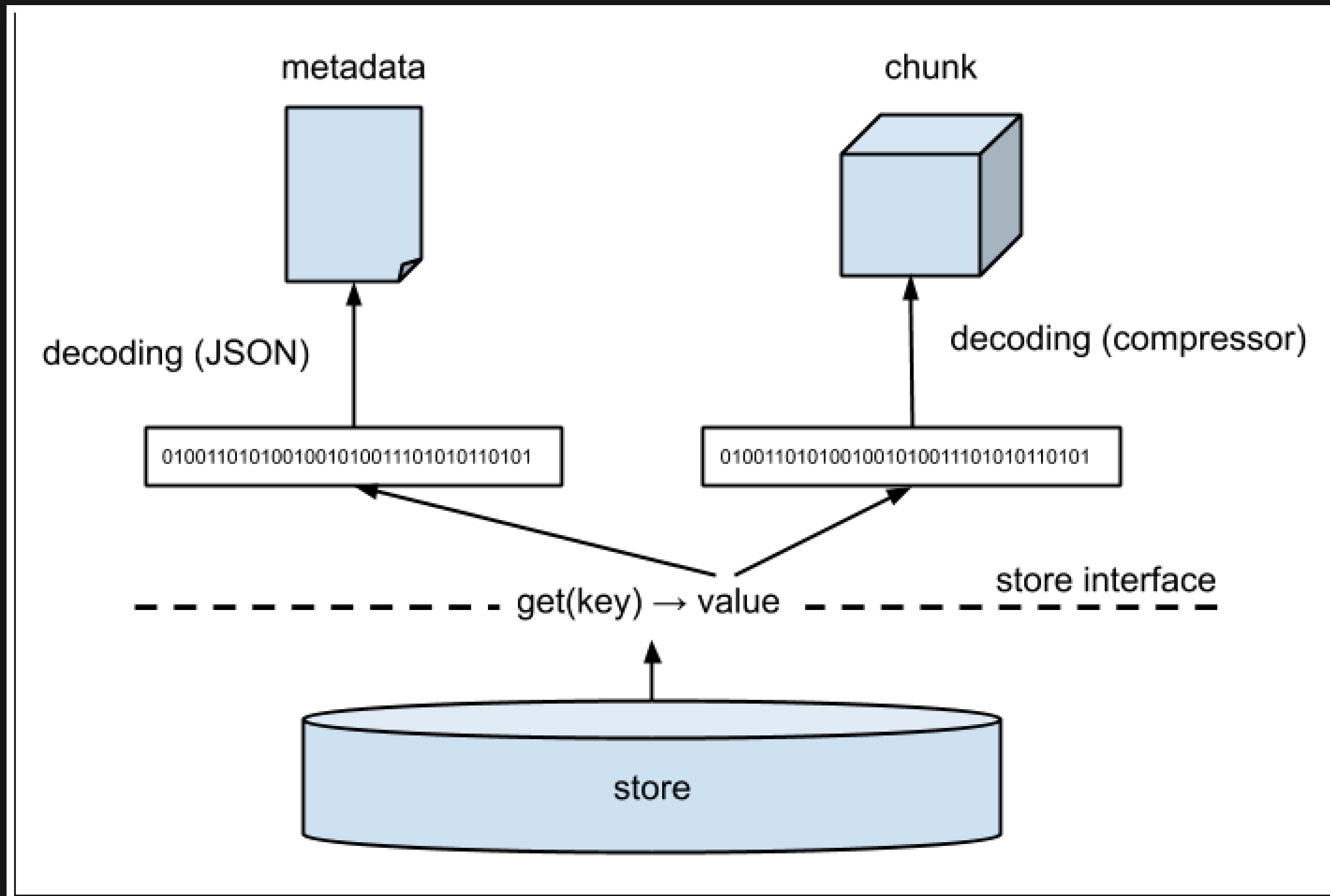
- If a store is **readable** it implements:
  - `get(key) -> value`
- If a store is **writeable** it implements:
  - `set(key, value)`
  - `delete(key)`
- If a store is **listable** it implements one or more of:
  - `list() -> keys`
  - `list_prefix(prefix) -> keys`
  - `list_dir(prefix) -> (keys, prefixes)`

Store interface - set(key, value)



N.B., implementing `set()` is optional; if not implemented, store is read-only.

Store interface - get(key) → value



Store interface - list operations

- List operations are required for discovering what groups and arrays a present in a hierarchy.
- List operations are optional; if not implemented, then the user/application has to find out what arrays and groups are present by some other means (e.g., via a protocol extension such as consolidated metadata or groups listing their children; or some out-of-band communication).
- Why are there three different list operations in the store interface? More on that later.

Store implementations

- As with current Zarr v2 and N5 protocols, goal of v3 store interface is to enable a range of store implementations using different storage technologies:
  - Local memory; file systems; zip files; local key-value databases (BDB, LMDB, Kyoto/Tokyo, LevelDB, ...); distributed key-value databases (e.g., mongo, redis, ...); relational databases; cloud object stores (S3, GCS, ABS, ...); ...
- Core protocol does not specify how to implement any of these. Leave that for **storage specs**.

Storage specs

- Each storage spec defines how the store interface is implemented in some concrete storage system.
- Generally will be obvious, but not always, and worth being explicit.
- E.g., expect there will be a file system storage spec, which defines the following mapping:

Operation	Implementation
`get(key) -> value`	read contents of a file
`set(key, value)`	write contents of a file
`list_dir(prefix)`	list contents of directory

Operation	Implementation
...	...

Storage protocol

- For each metadata document and chunk, need a unique **storage key.**
- In Zarr v2, storage keys formed like this, e.g., for 2D array at path "/foo/bar":
  - Array metadata document : "foo/bar/.zarray"
  - Chunk at grid index (0, 0) : "foo/bar/0.0"
- Two problems with the Zarr v2 protocol...

Zarr v2 problem 1 - race conditions

- When creating a node at some non-root path, e.g., "/foo/bar", then v2 spec says groups MUST be created at all ancestor paths.
- E.g., to create an array at path "/foo/bar", need to first:
    - Check if a group exists at path "/foo", if not then create it.
    - Check if a group exists at path "/", if not then create it.

Zarr v2 problem 1 - race conditions

- But if multiple arrays are being created in parallel, can lead to race conditions.
  - E.g., if arrays at "/foo/bar" and "/foo/baz" are being created in parallel, can get race conditions checking existence of and creating group at paths "/foo" and "/".
  - Further information here.
- v3 goal: avoid race conditions when creating nodes in parallel.

Zarr v2 problems 2 - inefficient

- Use case: user knows a group exists at path "/foo" and wants to know what children it has and for each child whether it's an array or sub-group.
- Use case: user wants a complete display of a hierarchy, i.e., some kind of tree view of all nodes and their types (array or group).
- With Zarr v2 protocol, both of these require a lot of calls to the store interface.
- On high-latency stores (e.g., cloud object stores) this can cause very noticeable delays.
- v3 goal: minimise store operations needed to explore/discover hierarchy.

Avoiding race conditions - implicit groups

- Zarr v3 currently avoids race conditions during node creation by allowing **implicit groups**.
- E.g., to create an array at path "/foo/bar", only a single store operation is required:
  - `set(array_metadata_key, encoded_array_metadata)`
- By creating an array at path "/foo/bar", groups at all ancestor paths ("/foo", "/") are **implicitly** created.

Hierarchy inconsistency?

- So, what happens if a user tries to do something that breaks the hierarchy model?
  - E.g., create arrays at paths "/foo" and "/foo/bar"? (So there is both an array and an implicit group at path "/foo".)
  - E.g., explicitly create both an array and a group at path "/foo"?
- Interesting question! Current v3 spec does not fully address this.

Hierarchy inconsistency?

- My current opinion: it's the user/application's responsibility to avoid hierarchy inconsistencies.
  - I.e., a Zarr core protocol implementation is **not** required to check or enforce hierarchy consistency during node creation (because this is hard to implement in a parallel/distributed context).
- If an application needs to enforce hierarchy consistency, then it could implement some mechanism for synchronising node creation, and/or it could implement some mechanism for checking consistency after node creation.

Making hierarchy discovery more efficient

- Current v3 resolves this by three means:

  1. Split the storage key space, so all metadata keys have prefix "meta/" and all chunk keys have prefix "data/".
  2. Change the format of metadata keys slightly (see below).
  3. Leverage the fact that many stores, including cloud object stores, can support `list_prefix(prefix)` and/or `list_dir(prefix)` operations, which natively list all keys with a given prefix.

v2/3 comparison - storage keys

Bootstrap metadata ...

Version	Storage key
v2	N/A
v3	`zarr.json`

v2/3 comparison - storage keys

Array metadata, e.g., for array at path "/foo/bar" ...

Version	Storage key
v2	`foo/bar/.zarray`
v3	`meta/root/foo/bar.array`

v2/3 comparison - storage keys

Group metadata, e.g., for group at path "/foo/baz" ...

Version	Storage key
v2	`foo/baz/.zgroup`
v3	`meta/root/foo/baz.group`

v2/3 comparison - storage keys

Chunk data, e.g., for chunk at grid index (0, 0) in 2D array at path "/foo/bar" ...

Version	Storage key
v2	`foo/bar/0.0`
v3	`data/foo/bar/0.0`
v3*	`data/foo/bar/0/0`

* chunk key format can be configured in array metadata

Core protocol - examples

- Initialize a hierarchy:
  - Perform `set("zarr.json", value)` where `value` is a serialised bootstrap metadata document.

Core protocol - examples

- Create an array at path "/foo/bar":
  - Perform `set("meta/root/foo/bar.array", value)` where `value` is a serialised array metadata document.

Core protocol - examples

- Create a group at path "/foo/baz":
  - Perform `set("meta/root/foo/baz.group", value)` where `value` is a serialised group metadata document.

Core protocol - examples

- List children of group "/foo":
  - Perform `list_dir("meta/root/foo/") -> (keys, prefixes)`.
  - Any returned key ending in ".array" indicates a child array.
  - Any returned key ending in ".group" indicates an explicit child group.
  - Any returned prefix indicates a child group implied by some descendant.

Core protocol - examples

- E.g., find all nodes in the hierarchy:
  - Perform `list_prefix("meta/") -> keys`.
  - All nodes (including implicit groups) and their types (either array or group) can be inferred from the returned keys.

Core protocol - examples

N.B., all of the above examples **required only a single store operation**. This is both a significant simplification and efficiency improvement over Zarr v2.

Discussion

- That's it for now.
- Comments/questions/discussion welcome via zarr-specs#16.
- Also please feel free to raise an issue on the zarr-specs repo, adding the "core-protocol-v3.0" label.