National Oceanography Centre

A digital solution to improve communication efficiency between environmental sensors and webservers (the osd2ERDDAP API)

Andy Matthews, John Walk, Lou Darroch, Jenny Brown,
Chris Cardwell, Tom Gardner, Owain Jones
and Margaret Yelland

2022

NOC Internal Document Number 24

A digital solution to improve communication efficiency between environmental sensors and webservers (the osd2ERDDAP API)

Purpose

The purpose of this report is to document the communications protocol and test system osd2ERDDAP developed as part of the Enhancing Climate Observations, Models and Data (ECO MAD) project. The approach allows deployed sensors to telemeter small quantities of arbitrary tabular Ocean Science Data (osd) directly to an ERDDAP server via the Internet in a way that is more efficient for the sensor than using ERDDAP's existing HTML Forms interface. ERDDAP is a data server that provides a simple and consistent way to download subsets of scientific datasets in common file formats and make graphs and maps. It is the online server used by the British Oceanographic Data Centre (BODC) and National Oceanic and Atmospheric Administration (NOAA) to publicly share environmental data in a format that meets international data management standards. Future use of this API with a range of ocean sensors and ERDDAP will increase the efficiency of data streaming. In turn this reduces power (and associated maintenance) requirements that is vital to deliver low-cost long-term monitoring networks, which support climate research and the management of climate impacts.

Contents

1 Background 3

2 Protocol 6

2.1 Transport & Application Layers 6

2.2 Security 6

2.3 HTTP Request Header 6

2.4 HTTP Response Header 7

2.5 HTTP Request & Response Bodies 7

3 ERDDAP Interface Application 9

3.1 ERDDAP Interface Data types 9

3.2 ERDDAP Interface Times 9

3.3 ERDDAP Interface NULL values 9

3.4 ERDDAP Interface Reserved Column Names 10

3.5 ERDDAP Interface Write Command (command ID = 0, version = 1) 10

3.6 ERDDAP Interface Read Command (command ID = 1, version = 1) 10

3.7 bmapping.json 11

4 Implementation 12

5 Future Improvements 13
5.1 Batched Commands 13
5.2 Extension of ‘time’ datatype 13
5.3 Scalability 13
6 Acknowledgments 14

1 Background

There is a common requirement for environmental sensors to send telemetry, typically tabular data, to a central server to allow key information such as sensor status, aggregated sensor data or early warning of changes in conditions to be retrieved remotely while they are deployed. Since power and bandwidth are typically at a premium in oceanographic deployments, the emphasis here is on small quantities of data being telemetered relatively infrequently by sensors with very limited resources in terms of processing power and working memory.

A requirement to be able to update a sensor's operating parameters remotely can also be of value for some deployments/situations. For that purpose, the scope of this project includes the *retrieval* of very small amounts of tabular data by the sensor when it makes occasional contact with the server. The scope does not include allowing the server to initiate communications with the sensor as the sensor is likely to be physically out of radio contact (e.g. submerged or with its communications systems turned off to reduce power consumption).

The aim of this project was to simplify the development and deployment of telemetry from the sensor's point of view.

This project was concerned with sensors connecting directly to the existing Internet infrastructure rather than peer-to-peer connections between devices for which Internet Of Things (IoT) messaging protocols like MQTT might be appropriate. Direct radio connection to the Internet is available via satellite systems such as Iridium RUDICS and Immarsat BGAN, and terrestrial 3G/4G/5G cellular (mobile phone) networks. Direct connection to these services can be seen as a heavyweight solution in terms of power, but for occasional telemetry the power requirements are reasonable and the solution has the attraction of simplicity: a sensor with this capability can be immediately deployed on its own anywhere in the world that a satellite or cellular network is reachable and only continued operation of that network is required for it to be able to communicate reliably.

Some commercial oceanographic vehicles (e.g. Liquid Robotics Wave Glider which uses Inmarsat BGAN and Kongsberg Seaglider which uses Iridium RUDICS) and some commercial loggers (e.g. Xylem Storm 3 Logger which uses the cellular networks) provide not only the Internet connection but also manage the communications between the vehicle/logger and their server, and host the data for you on their server. However, you usually pay for this service and integrating with these systems can sometimes (but not always) be more difficult than engineering the whole package yourself.

At the server end of things, the US National Oceanic & Atmospheric Administration (NOAA) developed the ERDDAP server application, which allows scientific data to be published for graphical display and downloads in various file formats via the World Wide Web. One ERDDAP *dataset type*, EDDTableFromHttpGet, allows for insertion as well as retrieval of tabular data using HTTP GET/POST requests over an Internet connection to the server with the data formatted as HTML forms. HTML forms are how humans send form data to web servers via Web browsers so it fits well with our requirement to send small quantities of tabular data occasionally. The server is typically set up to require secure HTTP (HTTPS) connections and the EDDTableFromHttpGet read/write requests additionally require a user name and password to provide some measure of security.

The viability of sensors inserting data directly to an ERDDAP server from the field has been demonstrated in the "Coastal REsistance: Alerts and Monitoring Technologies" (CreamT) project, which records and prototypes an early warning system for the overtopping of coastal sea-defences. Several ERDDAP datasets have been created by the British Oceanographic Data Centre (BODC) to receive and publish telemetry from the project's sensor arrays. The ERDDAP server is hosted at the

National Oceanography Centre in Liverpool (NOCL) accessed indirectly via a publicly visible Web server address. The CreamT sensors telemeter 1-100KB per telemetry session of tabular data every 10 minutes[1]. The connection to the ERDDAP server is made using off-the-shelf cellular modems on each sensor via a public 3G/4G (Three Mobile) phone network (see figure 1).

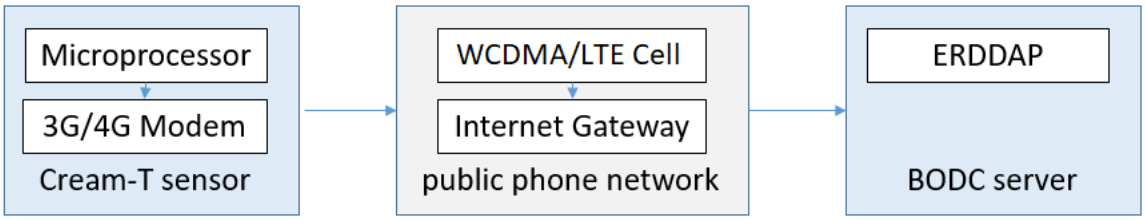

Figure 1. Transmission of data measured by a sensor to the server via the public phone network.

The 3G/4G cellular modem we used in this project was a SIMCOM 7600E. We deployed the sensors with a Waveshare evaluation board (£70 with aerials and leads) for the prototype but you can integrate the modem IC (£40 on its own) into your own electronics. That modem's firmware includes HTTPS support interfaced via text AT commands over a serial low-voltage UART interface to the sensor, a common arrangement with modems. The data costs for such low volumes of data over the phone network were tiny (£34/year for 12GB/12month Three Mobile Pay-As-You-Go SIM). The power requirements were manageable (3.6V 0.25A (continuous)/2A (max) power supply), about 2.5mAh per telemetry session (based on an actual average for this project of 35s per session of which most is a fixed connection-time overhead, the actual data transmission time is relatively small because the bandwidth is high). With 75 telemetry sessions per day, we were able to get about 50 days' telemetry out of a single non-rechargeable £15 SAFT LSH 2.0 3.6V LiPo cell (assuming 9Ah at 0.25A and 10°C). The CreamT sensors have 64KB RAM to run the entire firmware.

Whilst HTML forms are a well-proven and simple way to transfer data to a Web server, they are horribly inefficient in terms of the bandwidth used. The sensor firmware's RAM and CPU overheads in writing the HTTP requests or parsing their responses is also significant and that is far from ideal when these things are very limited.

Below is a mock-up of what must be transmitted by the SENSOR to ERDDAP to insert a single row into one of the ERDDAP dataset tables for the CreamT project when formatted as an HTML form:

```
GET
/erddap/tabledap/CreamT_nnnn_nnnn_nnnn.insert?stationID=D01&instrumentID=XXXXX&time=
1628693400&gpsTime=1628693402&wireID=1&sampleNUM10=123456&sampleNUM=123456&e
lVAR=1.23456&elMEAN=1.23456&elPTILE_6=1.23456&PTILE2_elVAR=1.23456&PTILE1_elVAR=1.2
3456&MINelPTILE_1=1.23456&MEDelPTILE_2=1.23456&MEDelMEAN=1.23456&THRESH1=10.00
&VOLT=7.2&SERNUM=1&TEMPH=22.5&calSCALE=1&calOFFSET=0&modOPTN=0&distP=0&rodTy
pe=0&dummy1=123456&dummy2=123456&dummy3=123456&dummy4=123456&dummy5=123
456&dummy6=123456&author=XXXXXXXXXXX_xxxxxxxxxxxxxxxx HTTP/1.1

Host:linkedsystems.uk
```

In this request more than half the characters are overhead, mostly in the column names preceding every value.

This is also a textual (ASCII-encoded) representation of the data and each of those characters counts one byte in the data transfer. The internal (binary) representation of data occupies a lot less memory, for example the integer value 1628693402 requires 10 bytes in its textual representation but only 4 bytes in its binary representation.

[1] For comparison of data volumes, 100KB is about one Web page with a couple of thumbnail images.

The conversion of binary values to their textual representation, particularly floating point numbers, presents both a RAM overhead (about 400 additional bytes of stack space) and a processing overhead for the sensor. It also risks losing precision where the internal representation is keeping more significant digits than the textual representation.

HTML Forms are not permitted to contain certain common punctuation characters, these must instead be encoded using multiple characters that are allowed. For example the square brackets required by ERDDAP to send array data must be encoded as three characters "%5B" instead of "[" and the plus character (such as in the scientific notation 3.1E+2) must be encoded as "%2B" instead of "+". The unpredictable expansion in this additional encoding present another problem for the sensor firmware which must pre-allocate sufficient memory for it (the use of dynamic memory allocation – as you might on a PC - is undesirable in embedded firmware)[2].

The total data expansion in Cream-T for telemetry was about 500%. It is this overhead which this Digital Solutions project sets out to address.

The primary intention of this solution was to design and test a more efficient interface between the sensors and the ERDDAP server for telemetering small amounts of data from the sensors to ERDDAP. Ideally this will be integrated with the ERDDAP server application itself in the future. However, the *protocol* defined by this project is sufficiently generic that it can also be used to make arbitrary RPC calls from the sensor to the server. This allows application-specific servers to be deployed for other projects using the same protocol, but still with the design goal of prioritising simplicity of implementation, and minimisation of RAM and processing requirements, at the sensor end.

In defining application-specific commands for this RPC interface, we aimed to be guided by the principles of the Representational State Transfer (REST) architectural model, but it was not a requirement of this project to enforce it (and even ERDDAP, by its use of HTTP GET for operations which modify (e.g. write and delete) datasets, is not following guidelines for using HTTP in a RESTful Web server).

The final API, osd2ERDDAP, was implemented as a Flask application within Python and deployed on the BODC infrastructure. Commands are sent by the sensor to osd2ERDDAP as binary messages, which translates the commands into the equivalent ASCII call to the BODC ERDDAP instance, and returns a message to the sensor indicating whether the command was successfully executed, as represented in figure 2.

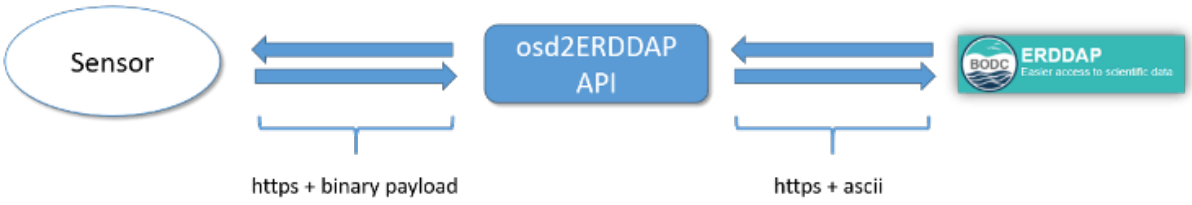

Figure 2. Message formats transmitted at each stage of the process.

This report documents the protocols the osd2ERDDAP API expects users to follow, gives an overview of the wider infrastructure required to deploy the API, and suggests some extensions that could be implemented in the future.

[2] Fortunately, ERDDAP seems currently to accept scientific notation without an explicit "+" character and the position of the square brackets in the request message is predictable making things a bit easier.

2 Protocol

2.1 Transport & Application Layers

We chose to use HTTPS. The underlying TCP/IP transport layer gives reliable transmission of packets so we do not need to deal with error checking, timeouts or retries of packet transmissions. The benefit of HTTPS is that there is good support for it already at both the sensor (modem) and server ends including the handling of the SSL/TLS security layer, and it is also typically easier to deploy than a bespoke TCP/IP application that would require special configuration of firewalls. The small cost is the need to generate and parse a small amount of text at the beginning of each HTTP message but we think the benefits outweigh that cost.

2.2 Security

The SSL/TLS layer encrypts communications between the sensor and server, and the server authenticates itself to the sensor by means of a certificate signed by a certificate authority recognised by the sensor. Currently the authentication is not mutual although this could be achieved by the sensor presenting a private key to the server: SSL and TLS both support this but it would require each sensor to be able to obtain a suitably signed private key and to be able both to store it and transmit it on request. For the time being security is limited to the unauthenticated sensor providing a password over the encrypted channel by means of the "Authorization" HTTP header field as described below, and in the case of ERDDAP reads and writes, by means of the ERDDAP "author" dataset column. Both passwords need to be made known to those configuring the sensors in advance of deployment, in some way considered to be secure enough.

2.3 HTTP Request Header

As a Web service, all communications between sensor and server consist of a *command* issued by the sensor (in an HTTP request) followed by a *response* (or batch of responses) issued by the server application (in the corresponding HTTP response)[3]. In the event of the sensor not receiving a response from the server it should close the HTTPS session and start a new one as this will ensure there is no confusion over late responses. Whether or not the command can then be safely retried is for the application to specify.

Like a Web service based on SOAP we use HTTP POST for our RPC requests but our payloads are binary, not XML. We use HTTP header fields only to inform the server which type/version of binary payload it is receiving and to send an authorization token. The request therefore look like this (where *application-path*, *version length*, *token* and *hostname* are replaced by real values as detailed below and <cl><lf> is a carriage-return line-feed sequence):

```
POST /application-path HTTP/1.1<cr><lf>
Accept: application/octet-stream; type=rpc.bodc/version<cr><lf>
Content-Length: length<cr><lf>
Content-Type: application/octet-stream; type= rpc.bodc/version<cr><lf>
Authorization: Basic token<cr><lf>
Host: hostname<cr><lf>
<cr><lf>
```

The *application-path* here is implementation defined but determines the application that will process this request (and therefore the command set available).

[3] Batches of commands and their corresponding responses are also supported, see section 3.6.

The application/octet-stream media subtype defines the "type" parameter as containing unspecified human-readable content. We require it to contain rpc.bodc/*version* which is the name and version number of the protocol defined by this document. The version number is an integer (1, 2, …) and the current version number is specified in section 3.5

Content *length* is the size in bytes of the request body that immediately follows this header. The exact format of the binary request body will be determined by the protocol version number (see section 3.5).

The authorization *token* is supposed to be an encoding of a "username:password" string in base64 but in practice the sensors will put whatever token they have been provided with here. Note that the user name and password are easily reverse-engineered so the use of an encrypted connection (HTTPS) is critical.

Hostname is the DNS name of the server (e.g. linkedsystems.uk).

2.4 HTTP Response Header

The response header informs the sensor whether or not the request was successful (at an HTTP level) with one of the codes 200 (success), 400 (invalid request) or 401 (unauthorized). The sensor may encounter other return codes all of which it should treat as an error. The sensor may also encounter header fields not described here (e.g. added by a proxy server) and these should be quietly ignored. The minimum response (with a *200* response in this example) looks like this:

```
HTTP/1.1 200 OK<cr><lf>
Content-Type: application/octet-stream; type= rpc.bodc/version<cr><lf>
Content-Length: length<cr><lf>
<cr><lf>
```

As with the request, the binary response message body is the *length* bytes immediately following this header.

2.5 HTTP Request & Response Bodies

The following data type names are used in the message body definitions.

Type Name	Type Description
uint8	1-byte unsigned integer
int8	1-byte two's complement signed integer
uint16	2-byte unsigned integer
int16	2-byte two's complement signed integer
uint32	4-byte unsigned integer
int32	4-byte two's complement signed integer
uint64	8-byte unsigned integer
int64	8-byte two's complement signed integer
float	4-byte IEEE 754 floating point number
double	8-byte IEEE 754 floating point number
time	4-byte unsigned integer storing seconds since Unix epoch (01/01/1970T00:00:00Z)
asciiz	ASCII characters in a fixed-width field, left-aligned and null-terminated
utf8	UTF-8 characters, in a fixed-width field

Square brackets after a scalar data type (integers, float, double or time) denote an **array** of that type (e.g. uint16_t[5] denotes an array of 5 x 2-byte unsigned integers). Square brackets after asciiz, utf8

or blob types denotes the **field width** (e.g. asciiz[30] denotes a left-aligned null-terminated ASCII-encoded string in a fixed width field of width 30 bytes).

The HTTP request and response bodies are each a **packed binary data structure** consisting of fixed header fields and an optional payload. The format of the header fields depends on the **protocol version** number in the HTTP request header which is currently at 1. The format of the payload (if any) depends on the **command ID** and **command version** number. The **byte ordering** specified in the command header applies to all multi-byte numbers in the command header and command payload. The server is required to be able to support both little-endian and big-endian ordering in the command and must use the same byte ordering in its response.

A command looks like this, where the data types are defined in section 4.

Field Name	**Data Type**	**Value**
flags	uint8	bit 0 = 0/little-endian, 1/big-endian, other bits unused
request id	uint8	request ID to be matched in response (see below)
command id	uint8	application-specific command id
command version	uint8	command version
command payload length	uint32	number of payload bytes (can be zero)
command payload bytes	uint8[]	payload bytes (command-specific)

A response looks like this:

Field Name	**Data Type**	**Value**
request id	uint8	request ID matching that in the command (see below)
response status	uint32	command exit status (command- specific)
response payload length	uint32	number of payload bytes (can be zero)
response payload bytes	uint8[]	payload bytes (command-specific)

The **Request ID** is an arbitrary positive number generated by the sensor that must be unique within the current HTTP request body only. It is only useful when sending batches of commands (see future enhancements section). The server must return the same Request ID in the response corresponding to a command. The special case of Request ID zero is used by the sensor to indicate that it does not require a response to a command: in that case the server must still return an HTTP response header but with no response body for that command.

Command IDs, version numbers and their payloads (if any) are application specific and determined by the server from the application-path in the HTTP request header. Commands for the ERDDAP Interface application (only) are defined in the next chapter.

3 ERDDAP Interface Application

The ERDDAP Interface application is a specific application of this protocol which allows sensors to write multiple rows of binary data to an ERDDAP dataset and to retrieve a single row of binary data. The dataset name, column names and types are not specified in the command payload, instead the payload contains a dataset ID which references an entry in a JSON file called bmapping.json on the server that specifies the mapping from the binary command payload to the ERDDAP dataset. The format of the mapping file is described at the end of this chapter.

3.1 ERDDAP Interface Data types

We have mapped the ERDDAP types to our own binary types as follows. As already explained, the sensor can use its native byte ordering for the multi-byte numeric types, the server will do any byte-swapping required if its native byte ordering is different.

ERDDAP Type	Type Name
ubyte	uint8
byte	int8
ushort	uint16
short	int16
uint	uint32
int	int32
ulong	uint64
long	int64
float	float
double	double
string	asciiz
string	utf8

3.2 ERDDAP Interface Times

ERDDAP stores times as seconds since the Unix epoch in columns of type double internally but returns them to queries as ISO 8601 formatted strings. The ERDDAP Interface Application hides this complication so that any fields declared as time in this interface will be transferred in the binary command and response payloads as 32-bit unsigned integers.

This implementation is dependent on ERDDAP returning ISO 8601 formatted strings given to the nearest second. For this reason, it is essential that the time_precision attribute for all time columns defined in the controlling datasets.xml file remains as the default value of 1970-01-01T00:00:00Z.

3.3 ERDDAP Interface NULL values

ERDDAP dataset rows can contain NULL values (i.e. empty columns). However, the returned binary message needs to return some fixed width value. Therefore, the ERDDAP Interface Application translates those values depending on their ERDDAP data type as follows:

ERDDAP Type	Null Value
all integer types	maximum value for that type (INTMAX)
floating point types	NaN
Strings	string full of zero-valued bytes
Time	zero (i.e. 1970-01-01 00:00:00Z)

If the table has been populated purely by use of the ERDDAP Interface Application, NULL values will not occur.

3.4 ERDDAP Interface Reserved Column Names

The following dataset column names (*not* data types) have special meaning to ERDDAP so if you transfer data to/from columns with these names make sure they have the same units and meaning. The binary data types shown below are appropriate but not mandatory:

Column Name	Data Type	ERDDAP required content
longitude	double	Longitude stored in degrees east
latitude	double	Latitude stored in degrees north
altitude	double	Altitude stored in metres with positive up
depth	double	Depth stored in metres with positive down
time	time	see above

3.5 ERDDAP Interface Write Command (command ID = 0, version = 1)

The write command allows multiple rows of data to be inserted into an ERDDAP dataset.

The command payload is a packed binary structure as follows.

Field Name	Data Type	Value
dataset_id	uint16	ID of dataset mapping in bmapping.json
author	asciiz[30]	ERDDAP author value
row_count	uint8	Number of rows to insert
dataset_bytes	uint8[]	Dataset bytes as defined by bmapping.json

The response payload is a packed binary structure as follows.

Field Name	Data Type	Value
row_count	uint8	Number of rows actually inserted

The command returns status 0 on success, 1 on all other errors.

3.6 ERDDAP Interface Read Command (command ID = 1, version = 1)

The ERDDAP Interface Read command allows a single row of data to be read from an ERDDAP dataset. The server returns the most recently inserted row (using the ERDDAP timestamp field) if the dataset contains any data.

The command payload is a packed binary structure as follows:

Field Name	Data Type	Value
dataset_id	uint16	ID of dataset mapping in bmapping.json

The response payload is a packed binary structure as follows.

row_count	uint8	Number of rows returned (0 or 1)
dataset_bytes	uint8[]	Dataset bytes as defined by bmapping.json

The command returns status 0 on success, 1 on all other errors.

3.7 bmapping.json

To allow arbitrary insert and read requests to be configured for multiple datasets, the mapping from binary request/response payload to ERDDAP EDDTableFromHttpGet request for each supported request is defined in an JSON file called bmapping.json that will be used by the server. The text encoding of the file will by UTF-8 (of which 7-bit ASCII is a subset). Each "dataset" object in bmapping.json defines the *dataset bytes* part of the binary command/response payload of the HTTP read/write request and the number of bytes is fully determined by the "dataset" object because all fields (even strings) have a fixed size. We define the JSON by an example and the notes that follow it.

```
{
  "datasets":
  [
    {
      "id": 1,
      "table": "CreamT_747f_b818_8edf",
      "columns":
      [
        {"name": "col_a", "type": "uint32"},
        {"name": "col_b", "type": "float", "precision": 5},
        {"name": "col_c", "type": "asciiz", "width": 20},
        {"name": "col_d", "type": "utf8", "width": 20},
        {"name": "col_e", "type": "blob", "width": 20}
      ]
    },
    {
      "id": 2,
      "table": "CreamT_747f_b818_8edg",
      "columns":
      [
        {"name": "col_a", "type": "uint32"},
        {"name": "col_b", "type": "uint32"}
      ]
    }
  ]
}
```

Notes:

- bmapping.json is stored within the osd2ERDDAP codebase, in the 'resources' folder. There is a separate version used within unit tests.
- Each server instance loads only a single bmapping.json file.
- Datasets defined in bmapping.json can be used for reads or writes.
- The dataset IDs must be unique in the file and therefore unique on the server.
- ERDDAP does not define true NULL data values (i.e. values to be omitted from analysis of the dataset) so columns omitted from write requests will receive their ERDDAP-assigned default values when the rows are inserted to the ERDDAP dataset.
- The "**name**" value is the ERDDAP dataset column name.
- The "**type**" value is one of the data types defined in section 3.5.
- The "**precision**" value is meaningful only on write requests with columns of type "float" and "double" and is the number of significant figures to output when converting the floating point value to text in the ERDDAP insert request. If omitted, the defaults are 8 for float and 17 for double.
- The "**width**" value is meaningful only with columns of type "asciiz", "utf8" and "blob" and is the size in bytes of these columns in the binary representation (they are always fixed width).
- The server should quietly ignore any values it does not recognise in the JSON file.

4 Implementation

The protocol has been implemented using the Python-based micro web framework Flask. The code is hosted in the NOC GitLab instance at https://git.noc.ac.uk/bodc/erddap-data-delivery/oceansensordata/osd2erddap-app (suitable permissions are required to access the code – contact BODC for details). The API has been deployed on BODC's external server at https://linkedsystems.uk/ocean-sensor-api/, although currently the API is configured to send and receive data from BODC's development instance of ERDDAP (https://dev.linkedsystems.uk/erddap/info/index.html) for final testing and while appropriate use cases are developed. The code is deployed from GitLab to the servers using BODC defined CI/CD pipelines.

At present, the authorization described in section 2.2 is carried out within the API itself, checking tokens belong to a given list, but ideally this would be implemented by the server itself to ensure nuisance messages are rejected at the earliest possible opportunity.

Messages from the sensor should be sent to the endpoint at the root URL of the directory. However, during development two extra endpoints were added for testing purposes – these have been retained as they have proved useful for diagnosing errors in messages sent by sensors. The valid endpoints and their purposes are as follows:

Endpoint	Description
https://linkedsystems.uk/ocean-sensor-api/	Main protocol – data updates are submitted to ERDDAP, body of HTTP response is binary described in 2.5
https://linkedsystems.uk/ocean-sensor-api/rawresponse	Updates are submitted to ERDDAP, but body of HTTP response is the message returned by the ERDDAP server
https://linkedsystems.uk/ocean-sensor-api/parser	ERDDAP not contacted, for push messages a JSON object representing the data to be submitted is returned, for pull messages the ID of the requested dataset is returned.

5 Future Improvements

The osd2ERDDAP API was developed as a proof of concept project providing a minimum viable project, so here we present some possible enhancements that could be added in the future.

5.1 Batched Commands

The time overhead in establishing a TCP connection (and deal with SSL/TLS) is problematic on a sensor which has limited power resources available for the modem. With HTTP/1.0 and HTTP/1.1 it is possible for the sensor to indicate via the Connection: keep-alive response header that it wishes the TCP connection to remain open for another HTTP request but whether or not that is respected depends on the server. For HTTP/2 that response header is forbidden although it may still be possible to configure the server to achieve the same keep-alive result. Failing that, the protocol supports the possibility of batching up commands to be carried out in the same TCP connection by putting an array of command requests, **each with a different Request ID**, in the HTTP request body. In that case the server is required to attempt all of the commands before returning an array of corresponding responses in the HTTP response body with matching Request IDs. The server can determine the number of requests using the Content-Length and protocol version from the HTTP request header, and the payload length of each command.

Batch commands were included in the initial project specification, hence the inclusion of the request ID in command requests, but implementation was dropped during delivery in order to focus on delivering a minimum viable product. Nevertheless, they should be relatively simple to add to the existing codebase – all that would be required is a function to establish whether an incoming request contains batched commands, a function to split the batch into individual request, and then code to create a suitable return message. Some care would need to be taken over how requests are sent to ERDDAP – if they cannot be entered in one call, protocols would need to be establish for how to handle cases where some of the ERDDAP calls were rejected.

5.2 Extension of ‘time’ datatype

ERDDAP stores time columns internally as double precision numbers representing seconds since the Unix epoch, but by default returns dates as ISO-8601 strings rounded to the nearest second. Therefore the ‘time’ datatype used within the osd2ERDDAP API.

However, the code could be extended to allow for time data of different precisions, allowing it to be used with sensors with sub-second sampling. This could be represented by adding a “precision” value to variables defined as “time” type in the bmapping.json file (see section 4.7). But the implementation would be dependent on the precisions defined in the bmapping.json file matching those defined by the time_precision attributes in the relevant ERDDAP dataset’s entry in datasets.xml.

5.3 Scalability

While the API was designed with small messages (<500kB) sent frequently for a small number of sensors, but in theory it should scale well. Messages are currently read into memory in their entirety. In theory this could cause problems should message size get extremely large. If necessarily the message parsing code could be adapted to stream the data, reading and processing one line of data at a time.

Each call to the API is handled in a single Flask thread, so extra capacity could be included by increasing the number of worker threads on the server. The limiting factor is likely to be the speed at which ERDDAP can update the underlying data storage.

6 Acknowledgments

This research and development project was the Digital Solutions task supported by NERC funding through ECO MAD (Enhancing Climate Observations, Models and Data). It utilised sensors and data transmissions made available through the "Coastal REsistance: Alerts and Monitoring Technologies" (CreamT) project, funded by the NERC Constructing a Digital Environment Strategic Priorities Fund (NE/V002538/1).