Network Tutorial

In this tutorial, a demo application is written on top of the network module.

The demo application is about a distributed library of books and book catalogues. The library consists of a set of books and catalogues. Each book has a name, year and rating. A book catalogue is a list of all recent books that were not included in the previous catalogue.

There is a single node responsible for the creation of catalogues and all the other nodes periodically publish new books. The goal is that all nodes in the system have the same set of books and catalogues.

The final code of this demo application can be found in `demo/library.py`.

Data Structure

In the first step we create the data structure and serialisation methods.

Datastructures that are used as `items` in the network are required to subclass the following two interfaces:

- `structures.ItemQualifier`
- `structures.ItemEncodable`

Our initial incomplete datastructures may look like the following:

demo/library.python

```python
from structures import ItemQualifier, ItemEncodeable

class Book(ItemQualifier, ItemEncodeable):

    def __init__(self, name, year, rating):
        self.name = name
        self.year = year
        self.rating = rating


class Catalogue(ItemQualifier, ItemEncodeable):

    def __init__(self, books):
        self.books = books
```

ID

What is missing are **qualifiers** also called IDs. That is because `items` in the network are required to have a qualifier that can be used to reference them. From a security standpoint it is also important that a qualifier is not **malleable** and also not **spoofable**. That is why we use a cryptographic hash function to create and check the qualifiers of each item.

demo/library.python

```
from struct import pack ①
import hashlib

def gen_book_id(book) -> str: ②
    m = hashlib.sha1()
    m.update(bytes(book.name, 'utf-8'))
    m.update(pack('i', book.year))
    m.update(pack('i', book.rating))
    hash_val = m.hexdigest()
    return f"book-{hash_val}"


def gen_catalogue_id(catalogue) -> str:  ③
    m = hashlib.sha1()
    for b in catalogue.books:
        m.update(bytes(b.id, 'utf-8'))
    hash_val = m.hexdigest()
    return f"catalogue-{hash_val}"


class Book(ItemQualifier, ItemEncodeable):

    def __init__(self, name, year, rating, id=None):
        self.name = name
        self.year = year
        self.rating = rating
        self.id = id

        if self.id is None:
            # Generate the matching book id.
            self.id = gen_book_id(self)
        else:
            # Check the id if it was already defined
            self.check_id() ④

    def check_id(self):
        if self.id:
            calculated_id = gen_book_id(self)
            if calculated_id != self.id:
                raise ValueError(f"ID check doesn't match the book content: "
                                 f"Given id: {self.id},"
                                 f"Calculated id: {calculated_id}")

    def item_qualifier(self): ⑤
        return self.id
```

```
class Catalogue(ItemQualifier, ItemEncodeable):

    def __init__(self, books, id=None):
        self.books = books
        self.id = id

        if self.id is None:
            # Generate the matching book id.
            self.id = gen_catalogue_id(self)
        else:
            # Check the id if it was already defined
            self.check_id()

    def check_id(self):
        if self.id:
            calculated_id = gen_catalogue_id(self)
            if calculated_id != self.id:
                raise ValueError(f"ID check doesn't match the catalogue content: "
                                 f"Given id: {self.id},"
                                 f"Calculated id: {calculated_id}")

    def item_qualifier(self):
        return self.id
```

① The pack library helps us encode integer values to byte array.

② Helper method that generates the id of a book using sha1.

③ Helper method that generates the id of a catalogue using sha1.

④ If the id is provided, check if the id matches the content of the book by recalculating it. For example, if someone sends us a book we deserialize it and accept it only if the book id is matching the content. Note that an adversary cannot spoof an ID because sha1 is collision resistant.

⑤ We also override the **required** interface method `item_qualifier` from `ItemQualifier` and return the book id.

Item Type

In the next step we create **item types**. Item types allow us to decode messages into the correct objects. It is required by any `ItemQualifier` to return the item type.

Item types are nothing but **unique integer values**. Unique in the sense that no other item from the same module or outside is allowed to have the same item type values.

Have a look at the built-in item type from the abc-protocol:

structures.python

```python
from enum import IntEnum
...
class ItemType(IntEnum):
    """
    Holds the three most iconic item types in abc protocol:
    TXN: Transaction
    ACK: Acknowledgement
    CHP: Checkpoint
    """
    TXN = 0xabce01 ①
    ACK = 0xabce02
    CHP = 0xabce03

    def __str__(self):
        return f'ItemType({self.name}, {self.value}'
```

① The transaction item type maps to the hex constant `0xabce01` which is simply the number `11259393`.

We create such an enum for library item types and return the correct item types in the `Book` and `Catalogue` classes:

demo/library.python

```python
from enum import IntEnum

...

class Book(ItemQualifier, ItemEncodeable):

    ...

    def item_type(self):
        return LibraryItemType.BOOK



class Catalogue(ItemQualifier, ItemEncodeable):

    ...

    def item_type(self):
        return LibraryItemType.CAT

...

class LibraryItemType(IntEnum):
    BOOK = 0xeee001
    CAT = 0xeee002
```

Serialisation - Transcription

Now we look at **serialisation** of our book and catalogue objects. **Transcription** refers to the process of transcribing items into network bytes So in this section we are going to add methods that take book or catalogue objects and write them as a **series of bytes** in a way that they can be parsed later again.

For this purpose the `transcriber.Transcriber` class exists. When transcribing, in the method `encode`, a `Transcriber` instance is given, and we use its method to encode the object's content as bytes.

demo/library.python

```
class Book(ItemQualifier, ItemEncodeable):

    ...

    def encode(self, transcriber):
        transcriber.write_text(self.id)
        transcriber.write_text(self.name)
        transcriber.integer(self.rating)
        transcriber.integer(self.year)

...

class Catalogue(ItemQualifier, ItemEncodeable):

    ...

    def encode(self, transcriber):
        transcriber.write_text(self.id)
        transcriber.integer(len(self.books)) ①
        for b in self.books:
            b.encode(transcriber) ②

...
```

① We need to write how many books are included in the message. We need this later when we want to reconstruct the catalogue object.

② Hand the transcriber to the book encoder and reuse our code.

Serialisation - Parsing

Now we look at how to parse network messages that contain book and catalogue items. That means we need to program a function that takes a `structure.Message` object and returns a list of item objects. Luckily, there is a helper class `transcriber.ItemsParser` in the `transcriber` module that allows us to concentrate solely on writing how to decode.

We provide a subclass of the `ItemsParser` that does this job. We receive an item type and a `transcriber.Parser` instance.

demo/library.python

```
from transcriber import ItemsParser

...

class LibraryItemType(IntEnum):
    BOOK = 0xeee001
    CAT = 0xeee002

class LibraryItemsParser(ItemsParser):

    def decode_item(self, item_type: int, parser: Parser) -> Any:
        if item_type == LibraryItemType.BOOK:
            _id = parser.consume_text() ①
            name = parser.consume_text()
            rating = parser.consume_int()
            year = parser.consume_int()
            return Book(name, rating, year, _id) ②
        elif item_type == LibraryItemType.CAT:
            _id = parser.consume_text()
            book_count = parser.consume_int()
            books = []
            for i in range(book_count):
                self.decode_item(LibraryItemType.BOOK, parser) ③
            return Catalogue(books, _id)
        else:
            return None ④
```

① Read the book content in the same order that we encode it

② Construct the book object and return it

③ Use this method to decode the books

④ Item type is unrecognized, so we drop it.

Now our datastructures are able to serialize as messages.

Item Container

When handling peer messages, we might be inclined to answer immediately, for example to request for an unknown item. This however might cause the network to be over-saturated by small messages.

Instead, it is advisable to use in-memory-buffers:

1. We put the set of unknown item ids in a special **set** that signals missing items.
2. If by chance, a network message delivers missing items content we will remove those from the set.
3. Eventually we ask our peers for an entire batch of books given their id.

Notice that when a peer responds to us by delivering the missing items, he will broadcast the item content. So many other network peers will receive the item content before they have even requested for it. If we are lucky each item will be requested exactly once.

We create a class that holds all the items in the system.

demo/library.py

```
from typing import Dict, Set

...

class LibraryContent:

    books: Dict[str, Book] ①

    cats: Dict[str, Catalogue] ②

    missing_items: Set[Tuple[LibraryItemType, str]] ③

    new_items: Set[Tuple[LibraryItemType, str]] ④

    requested_items: Set[Tuple[LibraryItemType, str]] ⑤

    def __init__(self):
        self.books = dict()
        self.cats = dict()
        self.missing_items = set()
        self.new_times = set()
```

① All our books are registered in this dictionary using their ID.

② All our catalogues are registered in this dictionary using their ID.

③ The set of all missing items. Instead of storing items in sets we will store their ID, because their ID already fully represents them:

④ The set of new items that we are interested in sharing with the network.

⑤ The set of requested items whose content, if we have them, we broadcast with the network.

Peer-To-Peer Communication

In this second part of the tutorial we look at how to accept, handle and send messages/ Each node is required to have the same set of books like all other peers. For this end, we use the items-broadcast mechanism to keep the set of books in sync. Among the message types supported by the abc network implementation we are interested at three message types:

- `structures.MsgType.items_content`: These messages contain a list of item contents. We can add missing books and catalogs to our library.
- `structures.MsgType.items_checklist`: These messages consist of a list of item ids. This way we

can check if there are any books or catalogues missing from our library.

- `structures.MsgType.items_request`: These messages request the content of some item given their ids.

We start by creating a `handlers.AbstractItemHandler` implementation. Look at `demo/library.py` for an example of how ab implementation can be done:

demo/library.py

```
class LibraryContent:

    books: Dict[str, Book]

    cats: Dict[str, Catalogue]

    missing_items: Set[Tuple[LibraryItemType, str]]

    new_items: Set[Tuple[LibraryItemType, str]]

    requested_items: Set[Tuple[LibraryItemType, str]]

    def __init__(self):
        self.books = dict()
        self.cats = dict()
        self.missing_items = set()
        self.new_items = set()
        self.requested_items = set()

    def cat_with_book(self, book_id):
        # Return true if at least one catalogue has a book with the given id.
        for cat in self.cats.values():
            for b2 in cat.books:
                if b2.id == book_id:
                    # The book is already contained.
                    return cat
        return None

    def find_content(self, item_type: int, item_id):
        if item_type == LibraryItemType.BOOK:
            if item_id in self.books:
                return self.books[item_id]
            else:
                cat = self.cat_with_book(item_id)
                for b2 in cat.books:
                    if b2.id == item_id:
                        return b2
        elif item_type == LibraryItemType.CAT:
            return self.cats[item_id]
        return None

    def has_content(self, item_type: int, item_id):
        if item_type == LibraryItemType.BOOK:
            return self.has_book(item_id)

        elif item_type == LibraryItemType.CAT:
            return self.has_cat(item_id)

        else:
            return False

    def has_book(self, book_id):
        # First check if the book is in a catalogue:
        if self.cat_with_book(book_id):
            return True
        # Lets look at loose books
        return book_id in self.books

    def has_cat(self, cat_id):
        return cat_id in self.cats

    def add_content(self, item_type: int, item_content):
        # Returns True if the content was added.
        if item_type == LibraryItemType.BOOK:
            return self.add_book(item_content)
        elif item_type == LibraryItemType.CAT:
            return self.add_catalogue(item_content)
        return False

    def add_book(self, book) -> bool:
        if not self.has_book(book.id):
            self.books[book.id] = book
            # New book added
            return True
        return False

    def add_catalogue(self, cat) -> bool:
        if not self.has_cat(cat.id):
            self.cats[cat.id] = cat
            # New cat added
            return True
        return False

    def mark_missing(self, item_type, item_id):
        self.missing_items.add(item_id)

    def mark_new(self, item_type, item_id):
        self.new_items.add((item_type, item_id))

    def mark_requested(self, item_type, item_id):
        self.requested_items.add((item_type, item_id))
```

```
class LibraryMessageHandler(AbstractItemHandler):

    def __init__(self, lc: LibraryContent):
        # Initialize AbstractItemHandler with the list of interesting item type and a
parser.
        super(LibraryMessageHandler, self).__init__([LibraryItemType.BOOK,
LibraryItemType.CAT],
                                                    LibraryItemsParser())
        # Global Set of Data
        self.library_content = lc
        # Timers for maintenance:
        self.timeout_timers = [
            (self.send_checklist, SimpleTimer(5.0)),
            (self.send_new, SimpleTimer(0.5)),
            (self.send_request_for_missing, SimpleTimer(0.5)),
            (self.send_content_of_requested, SimpleTimer(0.5)),
            (self.clean_books, SimpleTimer(10.0)),
        ]

    def handle_item_content(self, cs: "ChannelService", msg: Message, item_type: int,
item_content: Any):
        is_new = self.library_content.add_content(item_type, item_content)
        if is_new:
            # If it is a new entry, mark it so.
            self.library_content.mark_new(item_type, item_content)
        if item_content.item_qualifier() in self.library_content.missing_items:
            self.library_content.missing_items.remove(item_content.item_qualifier())

    def handle_item_request(self, cs: "ChannelService", msg: Message, item_type: int,
item_qualifier: str):
        # Mark that the item is being requested.
        # Broadcast the item at a later point.
        self.library_content.mark_requested(item_type, item_qualifier)

    def handle_item_checklist(self, cs: "ChannelService", msg: Message, item_type:
int, item_qualifier: str):
        if not self.library_content.has_content(item_type, item_qualifier):
            # Item from the checklist is missing.
            # Mark that it is missing so it can be requested at a later point.
            self.library_content.mark_missing(item_type, item_qualifier)

    def handle_item_notfound(self, cs: "ChannelService", msg: Message, item_type: int,
item_qualifier: str):
        # Ignore Item not found messages.
        pass

    def perform_maintenance(self, cs: "ChannelService"):
        for maintenance_method, timer in self.timeout_timers:
            if timer():
                # Timer of action has reached zero.
                # Perform maintenance action by calling the method that has the
```

```
action_name.
                maintenance_method(self, cs)

    @staticmethod
    def select_random_subset(super_set: List, subset_size=1000):
        if len(super_set) <= subset_size:
            return super_set
        random.shuffle(super_set)
        return super_set[:subset_size]


    def send_checklist(self, cs: "ChannelService"):
        checklist = list()
        # Add a subset of loose books:
        checklist += self.select_random_subset(list(self.library_content.books.values
()), 990)
        # Add a subset of catalogues:
        checklist += self.select_random_subset(list(self.library_content.cats.values(
)), 10)

        if checklist:
            # Broadcast checklist:
            cs.broadcast_channel().checklist(checklist)

    def send_content_of_requested(self, cs: ChannelService):
        requested_items = list(self.library_content.requested_items)
        # Only consider those items whose content is present:
        def content_of_requested_item_is_present(item_tuple):
            item_type, item_id = item_tuple
            if self.library_content.has_content(item_type, item_id):
                return True
            return False
        requested_items_id = list(filter(content_of_requested_item_is_present,
requested_items))
        # Only select 100 random items:
        requested_items_id = self.select_random_subset(requested_items_id, 100)
        # Retrieve the content of the requested items. Currently we only have a list
of item ids:
        def find_item_content(item_tuple):
            item_type, item_id = item_tuple
            return self.library_content.find_content(item_type, item_id)
        requested_items_content = list(map(find_item_content, requested_items_id))
        # Broadcast the content to the network:
        cs.broadcast_channel().items(requested_items_content)
        # From the set of requested items, remove those that we have sent now:
        for sent_item in requested_items_id:
            self.library_content.requested_items.remove(sent_item)

    def send_request_for_missing(self, cs: ChannelService):
        # Create a list of missing items:
        missing_item_list = list(self.library_content.missing_items)
        # Limit the amount of items that are sent, by picking a random subset:
        missing_item_list = self.select_random_subset(missing_item_list)
        if missing_item_list:
            # Broadcast the request of the list to the network:
            cs.broadcast_channel().fetch_items(missing_item_list)

    def send_new(self, cs: "ChannelService"):
        new_items = list(self.library_content.new_items)
        new_items = self.select_random_subset(new_items)
        if new_items:
            cs.broadcast_channel().checklist(new_items)
            # remove the mark of the items that we just sent
            for old_item in new_items:
                self.library_content.new_items.remove(old_item)

    def clean_books(self, cs: "ChannelService"):
        # Search all books that are currently loose but are also in a catalogue.
        duplicate_books = list()
        for book_id in self.library_content.books.keys():
            if self.library_content.cat_with_book(book_id):
                duplicate_books.append(book_id)
        # Remove the assembled list of duplicate books:
        for book_id in duplicate_books:
            del self.library_content.books[book_id]
```