ERRS: Project
Mini-Amazon / Mini-UPS

For this project, you will either be doing "mini-Amazon" (an online store) or "mini-UPS" (a shipping website). If you are doing Amazon, you will have to make your system work with the UPS systems in your **interoperability group** (IG)—2 groups doing Amazon and 2 groups doing UPS.

1 The "World"

Since you won't have access to real warehouses and trucks, your code will interact with a simulated world that I wrote. You will connect to the simulation server (port 12345 for UPS, port 23456 for Amazon), and send commands and receive notifications.

The server supports different worlds (identified by a 64-bit number). You may create as many worlds as you want. There is presently no authentication on the worlds, so please only use your own. You can use the `init-world` program to create a basic world to get started. Later, you can create your own worlds if you want.

Each world is comprised of a Cartesian coordinate grid where "addresses" are integer coordinates (so you will deliver a package to *e.g.*, $(2, 4)$). The world contains trucks (controlled by UPS) and warehouses (controlled by Amazon). These have to work together to deliver packages.

The messages you can send and receive are in the `.proto` files (amazon.proto and ups.proto) that I have provided. Notice that all messages either start with A or U to indicate which part they belong to.

The basic flow is that you send an A/UConnect message with the worldid that you want, and receive an A/UConnected response. Once you have received this response, you may send A/UCommands and receive A/Responses. You should not send any other message, nor expect to receive any—all of the details are embedded in the A/UCommands/Responses.

A/UCommands include two common options: simspeed and disconnect. You can adjust the simulation speed (higher numbers make things happen more quickly in the world). Note that the simulation speed is per connection, non-persistent, and only affects future events. If you set `disconnect` to true in a command, the server will finish processing whatever it is currently working on, then send a response with `finished = true`, and close the connection. Things will only happen in the world while you are connected to it (so you don't miss anything), however, if you just close the connection without asking, you might lose an in-flight message.

Note: simspeed is only for testing/debugging. You MUST NOT rely on a particular simspeed for the correctness of your program. When testing/debugging, if you want to try a large number of actions quickly, you might set it high. Likewise, if you wish to exercises particular timing-related conditions, you might set it slow. Your program MUST work correctly at ANY simspeed when the TAs use. They will have a version of the world

server which ignores simspeed commands that you send, and allows them to set the speed directly.

Amazon Commands details:

buy You can ask for more of a some products to be delivered to a warehouse. Specify item id, description (any text) and the quantity you want. If this product has never been seen before, it will be created. If the product has been seen before, you SHOULD provide the same description (if you use different descriptions for the same product id, the behavior is undefined). NOTE: buying new stock does not involve UPS.

topack Pack a shipment for delivery. You will be notified when it is ready. The warehouse that you request to pack the shipment MUST have sufficient inventory (and the inventory will be reduced accordingly).

load Load a shipment on to a truck. In order for this to succeed, the shipment MUST be packed (and you must have received a ready notification) AND the truck MUST be at the warehouse, ready to receive the shipment (the shipper must have sent them to pickup and they must have received notification of completion).

Amazon Response details:

arrived When you `buy`, you will later get a notification that your orders have arrived. At this time, you should update your records of what is in stock, and may use the goods described in this message to fulfill orders.

ready Notification that packing is complete

loaded Notification that you have finished loading a shipment onto a truck

UPS Command details:

deliveries Once a package has been loaded, you can issue this command to send the truck to deliver it to a particular location. Note that you MAY pickup other packages before making deliveries. Once the truck is sent to perform deliveries, it is busy until it completes them all, and cannot be given other commands (but other idle truck can). If you specify multiple deliveries at once, they will be performed in the order you list them in the command.

pickups Send a truck to a warehouse to pickup a package. The package need not be ready to issue this command. While the truck is en route, it is busy and cannot be given other commands.

UPS Response details:

completions You will receive this notification when either (a) a truck reaches the warehouse you sent it to (with a `pickup` command) and is ready to load a package or (b) a truck has finished all of its deliveries (that you sent it to make with a `deliveries` command).

At this point the truck may be given other instructions. Note that the completion tells you the current location of the truck. If you initialize a world yourself, you will also receive a completion notification with the initial position of each truck as soon as they finish initialization.

delivered You will receive this notification when each package is delivered.

Note that the world server's replies are **asynchronous**. You may send several requests, and receive the replies many **minutes** later. You should use appropriate identifiers in the responses to figure out what request a message is in response to. You also MUST NOT wait for the response to return a web page—if the response takes a few minutes, the browser will time out.

You MAY wish to separate the handling of world server communication from the handling of the web front end (hint: good idea). You could even go so far as placing the web server in a different Docker container from the daemon which interacts with the world server. In such a design, both programs can communicate through a common postgres database. You might even write these pieces of software in different languages.

1.1 Google Protocol Buffer Message Format

Because of some oddities of how GPB works, each message is preceded by a `Varint32` specifying its size in bytes. In C++ you would want to include

```
#include <google/protobuf/io/coded_stream.h>
#include <google/protobuf/io/zero_copy_stream_impl.h>
```

Then you could send a message with code like this:

```
//this is adpated from code that a Google engineer posted online
template<typename T>
bool sendMesgTo(const T & message,
                google::protobuf::io::FileOutputStream *out) {
  { //extra scope: make output go away before out->Flush()
    // We create a new coded stream for each message.  Don't worry, this is fast.
    google::protobuf::io::CodedOutputStream output(out);
    // Write the size.
    const int size = message.ByteSize();
    output.WriteVarint32(size);
    uint8_t* buffer = output.GetDirectBufferForNBytesAndAdvance(size);
    if (buffer != NULL) {
      // Optimization:  The message fits in one buffer, so use the faster
      // direct-to-array serialization path.
      message.SerializeWithCachedSizesToArray(buffer);
    } else {
      // Slightly-slower path when the message is multiple buffers.
```

```
      message.SerializeWithCachedSizes(&output);
      if (output.HadError()) {
        return false;
      }
    }
  }
  out->Flush();
  return true;
}
```

And receive a message with code like this:

```
//this is adpated from code that a Google engineer posted online
template<typename T>
bool recvMesgFrom(T & message,
                  google::protobuf::io::FileInputStream * in ){
  google::protobuf::io::CodedInputStream input(in);
  uint32_t size;
  if (!input.ReadVarint32(&size)) {
    return false;
  }
  // Tell the stream not to read beyond that size.
  google::protobuf::io::CodedInputStream::Limit limit = input.PushLimit(size);
  // Parse the message.
  if (!message.MergeFromCodedStream(&input)) {
    return false;
  }
  if (!input.ConsumedEntireMessage()) {
    return false;
  }
  // Release the limit.
  input.PopLimit(limit);
  return true;
}
```

2 Protocol Specification (10 pts)

Your IG MUST craft a protocol specification for how your servers will communicate with each other. This document MUST be submitted by 11:59 PM on Friday April 6^{th}. Your IG may use any reasonable communication protocol and data format you see fit. You SHOULD think carefully in designing this, and use RFC terminology (MUST/MAY/SHOULD) to be as exact as possible in constraining the behavior of the participants in this system.

Your IG MAY revise the protocol specification in the submission of your final project. If any revisions are required, you MUST leave deleted text in the document but color it red, and color any added text blue. If you replace one diagram with another, note the replacement in blue in the caption of the new diagram, and include the old diagram in an Appendix at the end.

The protocol specification will be graded on the following criteria:

- Clarity and precision: could your TAs or I implement a conforming server just by reading the specification?
- Conformance: does your code actually do what the (revised) specification says? (or did you just write some things and hack together a completely different piece of code)
- Completeness: did your initial document actually cover the behaviors you needed? (or did you find that you needed to heavily revise it)

We note that it is preferable (*i.e.*, you will receive a higher grade) if you find that your original document was not sufficient to accurately revise it, than to claim your original document was fine, but implement something different.

3 Bare Minimum Functionality (30 pts)

The first piece of functionality you should aim for is to be able to purchase an item, and have it go all the way through to delivery. Figure 1 illustrates.

For Amazon, this means that you need a web interface on which someone can buy a product (you don't need a catalog of products yet—you can start with just one "Buy" button and a fixed 'address' to deliver to). You then need to go through all the steps to get the package delivered (tell your warehouse to pack it, then when it is ready and a truck has arrived, load it).

For UPS, this means that you need a web interface which will display the shipments that exist, and their status (*e.g.*, created, truck en route to warehouse, truck waiting for package, out for delivery). Note that you will need to create a packageid (aka Tracking Number). These MUST be unique for the world. If you reuse a packageid in the same world, things will go wrong. You need to go through all the steps to deliver the package (send truck to warehouse, wait for Amazon to say its loaded, send it for delivery).

4 Actually Useful (40 pts)

Now that you have the basic functionality, its time to make it useful.

For Amazon, you should add the following features:

- A searchable catalog of products (you don't need a large quantity, nor real products).
- The ability to specify an address (*i.e.*, (x,y) coordinates) for delivery.

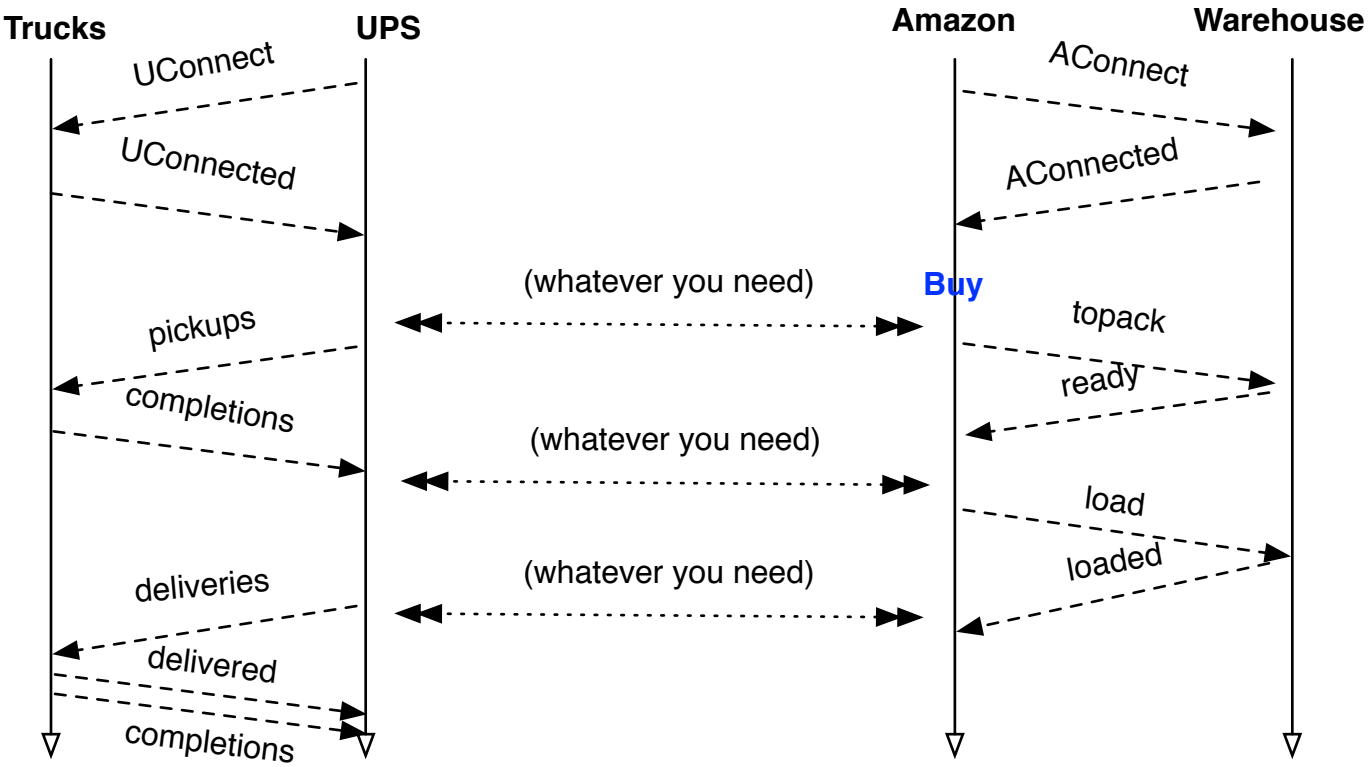

Figure 1: Bare Minimum Functionality Diagram. Note that Amazon¡-¿ UPS communication protocols are governed by the protocol document you write.

- The ability to specify a UPS account name to associate the order with (optional).
- Provide the Tracking Number for the shipment.

Note: there is no "payment" system—so you can just take a "credit card number" and pretend it is ok or just pretend that everything is free.
For UPS, you should add the following features:

- The ability to enter a tracking number and see the status of the shipment.
- User accounts (with user ids and passwords). If you are logged in, you should be able to do the following to packages you own (your user id was supplied to Amazon when the purchase was made):
  - See a list of all packages that belong to them.
  - See the details of the package (*e.g.*,items inside it)
  - If the package is not yet out for delivery, redirect it to a different address. (Note: if the user loses a race and the package goes out for delivery before you can update it, that is OK, but you need to tell them this).

5 Product Differentiation (20+)

At this point, everyone in the class has basically the same functionality. Now you need to differentiate your product ("store" or "shipping company") from the all the others out there—make yours the best in the class!

The last 20 points are flexible—you decide what features to add. Document them and justify them to the TA in a short writeup. It is possible to exceed 100 points by having a rich set of well-executed features. (However, it becomes exponentially more difficult to earn points the further above 100 you are).

You might consider features that require coordination between Amazon and UPS when deciding what to do here.

6 Other Notes

A few other notes:

- As usual, the TAs should be able to run your project with docker-compose up. You should make separate docker stacks for Amazon and UPS. The TAs will run docker-compose up for Amazon and separately for UPS. You MAY specify one place that a hostname needs to be changed in your docker-compose.yml files for this. In this setup, only one docker-compose.yml should run the world server, and the other should connect use the specified hostname to connect to it.

- You are not graded on UI/UX, however, we it must be at least usable. We recommend making it nice so you can show off your project (but save that for some final polishing). Make this project something you would be PROUD to show to potential employers, family, and/or friends.

- Communication between Amazon and UPS is entirely up to everyone in your IG. We make no requirements on the technologies/protocols/specifics used.

- You may use any language or combination of languages you want. However, you will probably want to use C, C++, Python, or Java (or Scala) to deal with the world server interaction, as those are the languages supported by Google Protocol Buffers.

- You should have at most ONE Amazon connection and at most ONE UPS connection to a given world at a time.

- I have given you a Dockerfile which will build an image that runs the world simulation. As noted earlier, it listens on port 12345 and 23456. You will need to set this up in your Docker Compose setup to have persistent storage for the database, and anything else you might want. If you just want to setup a simple world in it, you can use the provided `init-world` program, which will setup a simple world and give you its world id.

- For **debugging only** you could inspect the state of the world sim by entering the Docker container and examining the various tables in the `packagesim` database:

  deliveries These are the deliveries that trucks are currently doing (have been given a deliveries command, but not yet finished).

  shipments Shows all shipments that have been created, and whether or not they are completed. You MUST get to `completed = true` on your shipments before they are done.

  truckhas Shows the which shipments have been picked up by which trucks.

  trucks Shows the states of the trucks. 1 = idle, 2 = en route to a warehouse, 3 = invalid, 4= delivering, 5 = waiting for pickup. If statechange is not `infinity` then the truck will finish what it is doing at the given time.

  warehouse The position of each warehouse (by warehouse id: hid)

  whincoming When you purchase more items for a warehouse, they get listed in this table until they are delivered (artime shows when they will arrive).

  whready Shows what is ready at which warehouse

  whstock How much of which item each warehouse has in stock.

  Your programs MUST NOT directly examine or manipulate these database tables—they must interact with the world server through its GPB API.

- Also **for debugging/testing only** If you enter the Docker container, you will see two scripts: `rst.sh` and `restock.sh` scripts to reset the state of a world and restock it (respectively). Resetting the state will make all trucks idle, and clear all shipments, deliveries, warehouse ready state, etc. It will NOT clear warehouse stock or warehouse incoming stock. You can use this if you get your world in a messed up state. Restocking will add 10000 to every item in each warehouse that has fewer than 20000 (you can adjust these in the script).
- I wrote this in a day. It isn't so friendly when you do things wrong. It will give you an error message, but no emphasis was placed on error message friendliness. This got a morning of testing, so bugs are entirely possible—please let me know (and give me a test case to reproduce it) if you find one. I'll fix things and release new versions.

7 Deliverables

- Protocol document (Fri April 6).
- Code for your server (Amazon or UPS) (Fri April 27).
- A docker-compose setup which runs your software, as specified above (Fri April 27).
- A revised protocol document, showing any changes from your original (as described above) (Fri April 27).
- A writeup discussing your "product differentiation" features, and anything else that your group feels like needs to be discussed (Fri April 27).