ERSS: 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 provided for you. You will connect to the simulation server (port 12345 for UPS, port 23456 for Amazon), and send commands and receive notifications.

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

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. To create a new world for a pair of Amazon and UPS, UPS should send a `Uconnect` request without specifying a `worldid` number, so that the World Simulator would create one and return its ID in `UConnected` response. Both Amazon and UPS can create a new world and only when you want to create a new world do you leave the `worldid` blank.

Each world is comprised of a Cartesian coordinate grid where “addresses” are integer co-ordinates (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 basic flow is that you send an A/UConnect message with the `worldid` that you want and receive an A/UConnected response. Note only ONE Amazon and ONE UPS is allowed to connect to a world at the same. Upon successful connection, the `result` string in `A/UConnected` will be “connected!”, otherwise it will be an error message starting with “error:”. Make sure your `result` string is “connected!” before proceeding to any further actions. 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). Simulation speed has a default value of 100 and it's consistent once you specify it until you change it into another value. Note that the simulation speed only affects future events. If you set `disconnect` to true in a command, the server will finish processing whatever it is currently working on (your current `A/UCommands`), then send a response with `finished = true`, and close the connection.

`A/UCommands` and `A/UResponses` also implement `ack` numbers to avoid losing an in-flight message. The ack mechanism works as follows. For each request inside `A/UCommands`, there's a `seqnum` (you should keep track of the incrementing of `seqnum` coming from your side). When World Simulator receives commands from your side, it will check the `seqnum` of each request. Then it will process the request and return responses with `acks` of those seqnums. The same thing happens when World Simulator send you responses, which means if you don't return `ack`, World Simulator will send the same responses for multiple times. Don't assume World Simulator receives all of the requests in your `A/UCommands` until you receive those `acks`.

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 exercise 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: (note all commands include a sequence number for acknowledgement as described above)

buy You can ask for more of 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 ware- house 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).

queries ask the status of a package by specifying the packageid. Note you can do query at any time.

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

packagestatus tell the current status of one package that you queried. Possible package status: packing, packed, loading, loaded, delivering, delivered.

error indicates that you failed to meet any of the MUST requirements specified at "Amazon Commands details" above. Read the err string carefully for more information.

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. You MAY send more deliveries requests while the truck is delivering other packages. You MAY even change the destination of a package by sending a delivery request again before it arrives its destination. World Simulator allows idle truck carrying undelivered packages. 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 pick up a package. The truck need not have an "idle" status; it can also have an "arrive warehouse" or "delivering" status. If a truck receives pickups requests in the middle of a delivering, it will immediately quit the

current delivery and turn to the specified warehouse. Later whenever the truck has a "delivering" status again, it always starts from where it quits. The package need not be ready to issue this command. While the truck is in route, it is busy and cannot be given other commands.

queries ask the status of a truck by specifying truckid. Note you can do query at any time.

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.

delivered You will receive this notification when each package is delivered. Note that when each package is delivered, a `delivered` response will be sent. When all deliveries are finished, you will receive a `completions` response.

truckstatus tell the current status of a truck that you queried. Possible truck status: idle, traveling (when receives pickups requests and is on its way to warehouse), arrive warehouse, loading (loading package, after loading package finish, go back to "arrive warehouse" status), delivering (when finished all deliver job, go to status "idle").

error indicates that you failed to meet any of the MUST requirements specified at "UPS Commands details" above. Read the err string carefully for more information.

Note World Simulator has a time-out value set to be 10 mins, if you found that you lose the connection, don't panic, just connect again. Also 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 adapated 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 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 adapated 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 Thursday, April 7. 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.

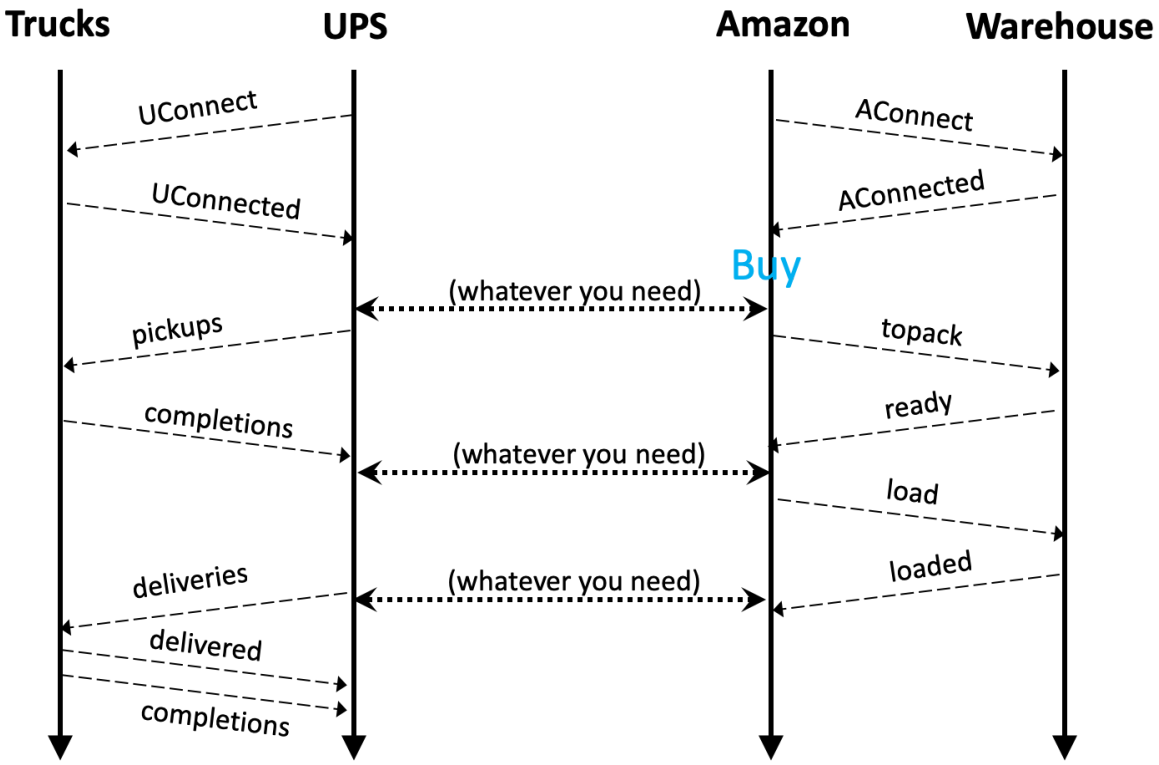

Figure 1: Bare Minimum Functionality Diagram. Note that Amazon-UPS communication protocols are governed by the protocol document you write.

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 it's loaded, send it for delivery).

4 Actually Useful (40 pts)

Now that you have the basic functionality, it's 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 check the status of an order.
- The ability to specify an address (i.e., (x,y) coordinates) for delivery.
- 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.
- You will be given 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 `worldSim` database:

```
            List of relations
 Schema |            Name             | Type  |  Owner
--------+-----------------------------+-------+----------
 public | apm_res_helper              | table | postgres
 public | apurchasemore_response      | table | postgres
 public | completion_response         | table | postgres
 public | error_response              | table | postgres
 public | finish_response             | table | postgres
 public | incoming_msg                | table | postgres
 public | outgoing_msg                | table | postgres
 public | package                     | table | postgres
 public | package_delivered_response  | table | postgres
 public | package_product             | table | postgres
 public | pending_query               | table | postgres
 public | product                     | table | postgres
 public | rl_response                 | table | postgres
 public | truck                       | table | postgres
 public | warehouse                   | table | postgres
 public | world                       | table | postgres
```

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** only, when you connect to the world's database you may clean the tables or make other queries / modifications.
- The worldsim may not be fully friendly when you do things wrong. It will give you an error message, but no emphasis was placed on error message friendliness. This has some basic testing, but bugs are entirely possible – please let us know (and give a test case to reproduce it) if you find one. We'll work to fix things and release new versions.

7 Deliverables

- Protocol document (Thurs Apr 7).
- Code for your server (Amazon or UPS) (At the project due date).

- A docker-compose setup which runs your software, as specified above (At the project due date).
- A revised protocol document, showing any changes from your original (as described above) (At the project due date).
- A writeup discussing your “product differentiation” features, and anything else that your group feels like needs to be discussed (At the project due date).