EECS 489 PA1: Peer-to-Peer Search

This assignment is due on Friday, 30 Jan 2015, 10 pm.

Preamble

Review the grading policy page on the course website. Remember that to incorporate publicly available code in your solution is considered cheating in this course. To pass off the implementation of an algorithm as that of another is also considered cheating. For example, if the assignment asks you to implement sort using heap sort and you turn in a working program that uses insertion sort in place of the heap sort, it will be considered cheating. If you can not implement a required algorithm, you *must* inform the teaching staff when turning in your assignment by documenting it in your writeup.

Graded Tasks (100 points total)

In this assignment you are to build a peer-to-peer (p2p) network and perform a search for an image on the p2p network.

1. Implement a peer node similar to the one you implemented for Lab 2. You may re-use code from Lab 2 (20 points)
2. Make the peer table size a run-time parameter (5 pts)
3. Return multiple known peers, up to a maximum number (10 pts)
4. Automate peer join and redirection (15 points)
5. Client image query, adapted from Lab 1. You may re-use code from Lab 1 (15 points)
6. Search for an image (35 points)
7. Writeup

Your Tasks

1. A Peer Node

Your first task is to write a peer node. If you've implemented Lab 2 and have decided to build this assignment on top of your working Lab 2, you're done with the first task of this assignment. If you have not implemented Lab 2, you may want to review the support code and specification of Lab 2. They go into more details and guide you step by step on what needs to be implemented. In the remainder of this document I will assume that you are familiar with the Lab 2 specification and support code.

To boot strap the p2p network, we first start a peer by itself. When a peer is started without being given another peer to connect to, it simply creates a socket and listen on it for incoming connections. Everytime a peer starts, we also have the peer prints to screen/console its fully qualified domain name (FQDN), and the port number it is listening on. Subsequent peers are then started with the FQDN:port of the first peer. Your code must take an optional command line option `-p FQDN:port` as in Lab 2. When a peer is given the FQDN:port of another peer at start time, it tries to join that peer in the p2p network by creating a socket and connecting to the peer.

A peer that receives a join request will accept the peer if and only if its peer table is not full. Whether a join request is accepted or not, the peer always sends back to the requesting peer the FQDN:port of at

least one peer in its peer table to help the newly joined peer find more peers to join.

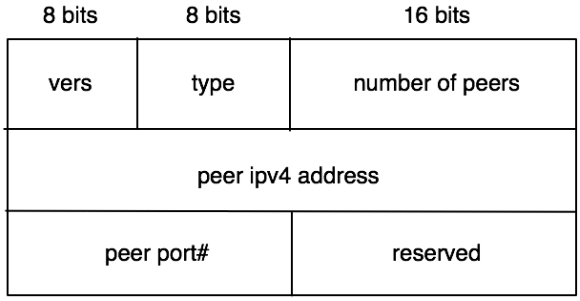
In this assignment, we assume that once a peer joins the network, it never leaves the network. So you don't have to worry about cleaning up after departed peer. You must, however, ensure that none of the peers crash when one of them leave, so you can take down the network one peer at a time without the others crashing.

2. Larger Peer Table

The support code for Lab 2 makes two simplifying assumptions: (1) that the peer table of each peer can hold only 2 peers and (2) that the acknowledgement message sent back to a joining peer contains at most 1 alternate peer. Your second task is to make the peer table size a run-time parameter. Add an optional `-n maxpeer` option to the command line argument so that user can specify the peer table size at run time. A study of the Gnutella p2p network found that half of Gnutella peers supports at most 2 peers. Even though there are peers that support over 130 other peers, the mean number of peers supported is 5.5. If the `-n` option is not specified in the command line, use a default value of 6 (`PR_MAXPEERS`). If it is specified, the provided number must be at least 1. Given the small number of peers expected, you can implement the peer table using a simple table or list with linear insert and/or search times. It may be useful to implement the peer table as a C++ class. You may use STL to implement the peer table. If you build off `peer.cpp` from Lab 2, this task shouldn't amount to more than about 16 lines of new and modified code.

3. More Peers

Your third task is to support more than 1 returned peer with each join acknowledgement message. The acknowledgement message **MUST** be of the following format:



where `vers` must have the value `PM_VERS` and `type` must be `PM_WELCOME` or `PM_RDIRECT`, all as defined in Lab 2. The field "`number of peers`" must contain the exact count of the number of peers returned (starting from 0). Peer addresses and port numbers (and reserved field) subsequent to the first one simply follow the first one in the byte stream. So each peer takes up 64 bits on the returned packet, including the reserved field for each peer. The number of peers returned **MUST** be ≤ 6. If your peer table holds more than 6 peers, you send only (the first, the last, or random, your choice) 6 peers. Unlike in Lab 2, when the number of peers is 0, the acknowledgement packet **MUST** consist only of the first 32 bits of `pmsg_t`, i.e., without any `peer_t` attached. This task should take about 36 lines of modified and new code.

Note the bolded **MUST**s above. Whenever you see a **MUST** in a protocol specification, you **MUST** follow it to the letter, to ensure that your code can interoperate with other implementations. In this case, your code must interoperate with our peer code for grading purposes. If you don't follow the packet format to the letter, your code will not interoperate with our code and you will get zero points. Also don't forget to use `ntohs()` and `htons()` where necessary. A reference peer (`refp2pdb`) that runs on CAEN

eecs489 hosts (`caen-eecs489p01.engin.umich.edu` up to `p04`) is available in `/afs/umich.edu/class/eecs489/w15/FILES/`. You can test your implementation against the reference implementation to ensure interoperability. The reference implementation only runs on GLU/Linux, so don't try to run it on Debian, Ubuntu, Mac OS X, or Windows machines, including the ITCS and other CAEN machines. Remember that you can connect to the CAEN eecs489 hosts only through UMVPN and MWireless or from CAEN Lab desktops.

4. Automatic Join

In Lab 2, when a peer receives a `PM_RDIRECT` message, it simply prints out a join failure/redirection message to the console. It is then up to the user to re-run the peer to join another peer. Your fourth task is to automate this process. When a peer recieves a `PM_RDIRECT` message, instead of simply printing out a redirection message, your code should go down the list of returned peers and try to join each one of them until you have filled up your peer table. Actually, you need to do this even if you receive a `PM_WELCOME` message if your peer table is not yet full. If your peer table is full but the list of peers returned to you by the peer you try to join is not yet exhausted, even if some peers in the table are still in "pending" state and may end up rejecting you, you can just throw away the remainder of the list. If your peer table is still not full after you've exhausted the list of peers, try to join with peers subsequently referred to you by the peers you contacted. You need to keep track of four cases: (1) you don't want to join again with peers who are already in your peer table, (2) you don't want to initiate another join with peers you already have a pending join, (3) you don't want to join again those peers from which you have received a `PM_RDIRECT` message, and (4) if you try to join a peer at the same time it tries to join you, only one of you will successfully form a link.

The first case is easy to check for: just make sure the peer you want to join is not already in your peer table. For the second case, if you enter into your peer table all your pending joins, then this case reduces to the first case. You may want to add a "pending" field to your peer table entry so that you don't forward a search packet (see next task) to pending peers. For the third case, you need to keep a separate "peering declined" table. You are required to check against the last `PR_MAXPEERS` that have declined to peer with you. So your "peering declined" table which could simply be a circular array of `peer_t`. Prior to attempting to join with a peer, you check against this table just like you would against the peer table. If you received a `PM_RDIRECT` from a peer, close the connection and move the peer from your peer table to your "peering declined" table. Otherwise, clear the peering table entry's pending field. If your peer table is full, even if some of the entries are still "pending", you don't initiate another join. (This also serves as a control to make sure that you don't flood the network with join requests!) As for the fourth case, only one of the two `connect()` attempts will succeed. The other will return with an error and the system `errno` variable will be set to `EADDRNOTAVAIL`. In which case, simply clear the peer table entry of the failed connection.

If you've exhausted the returned peer lists and all peers are already in your peer table or peering declined table and your peer table is still not full, just chill out and do nothing and wait for new peers to connect to you.

To bind the same address and port to multiple sockets, on MacOS X and Windows, it is usually sufficient to set socket option `SO_REUSEADDR`. But on Linux, you would need to set socket option `SO_REUSEPORT` in addition.

WARNING: don't confuse yourself by implementing the peer code as a multi-threaded process. With multithreading, you'd have to serialize access to the two tables. Just use the single-threaded event-driven model with `select()` as in Lab 2. Then you'll be dealing with only one message at a time and don't have to worry about inconsistent states caused by multiple messages arriving at the same time. This task

shouldn't take more than 35 modified and new lines of code.

5. Client Image Query

Your next task is to integrate the image query from Lab 1 with the peering code from Lab 2. If you have implemented Lab 1, you can re-use your code. If you have not implemented Lab 1, you want to review its support code and specification to complete this task. The client from Lab 1, `netimg` should work with one line of modification: the query packet now has an additional field, `iq_type` which must be set to `NETIMG_QRY` in `netimg.cpp:netimg_sendqry()`. A new `netimg.h` with the definition of `NETIMG_QRY` and an updated `iqry_t` is available for download. With the change in packet format, `NETIMG_VERS` has been incremented so that you won't accidentally use any old `netimg` binary with the new server.

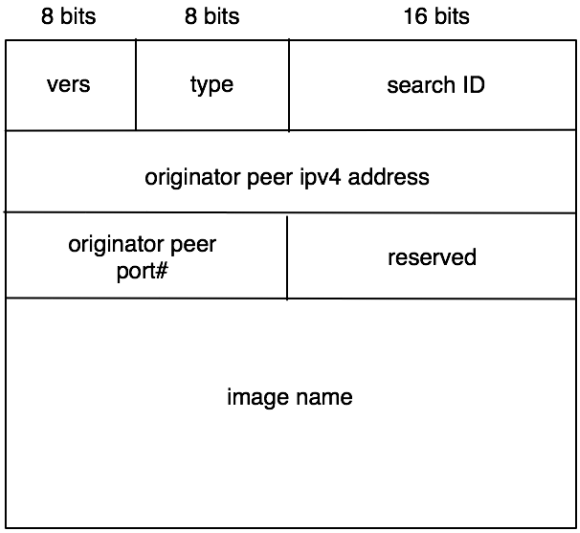
Next, incorporate the server code, `imgdb`, into the peer code, which we will then call `p2pdb`. The server will use two different ports: one to handle peer-to-peer network maintenance traffic and another to handle image query traffic. In addition to initializing the socket to handle peering traffic (as is done in `main()` of `peer.cpp`), you now need to create and initialize a socket to handle image traffic (simply by calling `imgdb_sockinit()`). We will call the former *peer socket* and the latter *image socket* henceforth. You need to register this image socket with `select()` along with all the other peer sockets. We will use one image socket for both client query and peer image-search reply (next task)—which, incidentally, is the reason for the `iq_type` field in `iqry_t` packet.

When a client queries for an image, the server first searches its own database (so to speak :-) for the requested file name, by calling `imgdb_loadimg()` as in Lab 1. If the image is found, it is returned to the client and the connection is then closed. If the image is not found, the server checks whether it is already searching for an image in the peer-to-peer network on behalf of another client. If so, it returns an `imsg_t` packet to the new client with the `im_found` field set to `NETIMG_EBUSY`. That is, a server performs only one peer-to-peer search at any one time. You have to decide how to determine that a server is already serving another client. You may update the error reporting in `netimg.cpp` to handle this new error type, but you're not required to do so, i.e., it's ok if your `netimg` client reports that image is not found when the server is busy. We will discuss how to handle peer-to-peer search in the next task.

At this point, you should test your code and verify that your `netimg` client and `p2pdb` server work as in Lab 1 to serve up image files that are local to the server. To build the client, you'll need the files `netimg.cpp`, `netimglut.cpp`, and `netimg.h`. To build the server/peer, you'll need the files `peer.cpp`, `imgdb.cpp`, `netimg.h`, `ltga.cpp`, and `ltga.h`. See the provided `Makefile`. On Windows, you'll need `wingetopt.c` and `wingetopt.h` also. This task should take about 26 lines of modified or new code in `peer.cpp` and about 1 line of code in `netimg.cpp`. You'll need to comment out the `main()` function in `imgdb.cpp` also.

6. P2P Search

When a peer cannot find an image locally, it sends out a search packet through the peer-to-peer network. When a queried image is found, the peer holding the image connects directly with the peer searching for the image (*orignating peer*) and transfers the image to the originating peer, who then forwards it to the client. As explained in the previous section, this connection is made to the originating peer's *image socket*. Thus the search packet must contain the originating peer's address and its image socket's port number, along with the name of the image being searched for. You may re-use code from Lab 2 for this task. The query/search packet **MUST** follow this format:



where `vers` **MUST** be `PM_VERS` as before, `type` **MUST** be `PM_SEARCH` (= `0x4`). The "`search ID`" field is a way for you to differentiate subsequent searches for the same image name (see below). It can be a simple monotonically increasing number, incremented for each search. You don't have to worry about the number wrapping around in this assignment. The "`peer port#`" and "`peer ipv4 address`" are the IPv4 address of the originating peer and the port of the *image socket*, NOT the *peer socket*. Since the *image socket* will be bound to `INADDR_ANY`, you can't use `getsockname()` to find out the address of the socket (it will return 0's). One way to obtain the address of the current host is to call `gethostname()` and then call `gethostbyname()` to obtain the address of the host. You may want to modify `imgdb_sockinit()` to return you the address and port number of the newly initialized socket. Don't forget to use `htons()` and `ntohs()` where appropriate. The peer initiating an image search sends a copy of this search packet to all the peers in its peer table.

Search packets are sent along the connections made between peers, i.e., the "links" forming the p2p network. Peering relationships that are still "pending" (see the "Automatic Join" section above), should not be used to forward search packet. Once you have sent out a search packet to all your connected peers, you don't need to send it again if new peers connect to you at a later time. When a search packet arrives at a peer, the peer must check whether it has seen the same query before. You don't have to keep a very long history. Just keep the last `PR_MAXPEERS` number of the most recent searches and check against them. Again, you can keep these in a circular array. If the peer has seen the search in the recent past, it simply drops the packet. Otherwise, it checks whether it is holding a copy of the queried image (by calling `imgdb_load()`). If it does not have a copy of the image, the peer forwards the query further to all its peers, except for the peer whence the query arrives~~ and the originating peer~~. Your code must be able to make these determinations and not forward the search packet in the ~~four~~ three cases mentioned here. You *will* be deducted points if your queries loop on your p2p network because your node doesn't drop duplicate queries.

If the peer has no other peer to forward the query to, it simply drops the query. If the peer does hold the image, it creates a new socket and connects to the querying peer at the address and port number listed in the search packet. Thus the image is not transmitted on the "links" of the p2p network, but on a separate connection created just to transfer the image. Once the image transfer is completed, the connection is closed by the peer initiating the transfer. The originating peer then forwards the image to the client requesting it and closes the connection to the client. If the originating peer receives multiple copies of the requested image, it only returns one copy to the client. If it receives an image when it is not waiting for any search reply, it can simply closes the connection with the peer (if you're using `imgdb_sendimg()`, you'd need to modify it to be more forgiving to closed connection and not crash your peer unnecessarily). At any one time, a peer can only perform a search on behalf of one client. Your code should enforce this.

If a reply for an old search arrives after a new client initiated a new search, the peer will return the wrong image to the new client. You **don't** have to worry about this error case.

Image transfer between peers and between a peer and its client MUST follow the same protocol as in Lab 1: you MUST precede the image with an `imsg_t` packet.

The `type` field must be set to `NETIMG_RPY`. You can use `imgdb_sendimg()` to perform all image transfers. You may want to modify it such that image transfer between peers is done fast, as one segment. You may also want to modify its argument so that it takes a "`char *`" instead of "`LTGA *`". (You can obtain the image in the LTGA container by calling `LTGA::GetPixels()` method.)

Since a search may fail to find the queried image, the querying peer must set a timer, as the last argument to `select()`, after which it gives up waiting for a reply, informs client that the image cann not found, and closes the connection to the client. You can use 1 second timeout value. Since the timeout can be interrupted by activities in the other sockets you're selecting on, you'd normally compute how much time has passed and reset the timeout to the smaller time value in your subsequent call to `select()`. To keep things simpler for you, you can continue to use 1 second time on each call to `select()`. Your peer-to-peer network is not so busy that this will lead to indefinite timeout.

You will notice that on the peer socket, a peer could receive a join acknowledgement packet or an image search packet. While on the image socket, a peer could receive a client query packet (`iqry_t`) or a search reply packet (`imsg_t`). The common denominator for all these packet types are the first two bytes. You can grab the first two bytes of a packet, check that it is of the expected version number, then decide how to receive the rest of the packet by the type encoded in the second byte.

This task takes 100 to 115 lines of modified and new code.

Testing Your Code

You will be graded for correctness primarily by running your program on a number of test cases. If you have a single silly bug that causes most of the test cases to fail, you will get a very low score on that part of the programming assignment *even though you completed 95% of the work*. Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. Each testcase should test only one particular feature of your program. Just as Apple, Google, Microsoft, etc. do not ask for testcases from their customers prior to releasing their code, it is your responsibility to test your code thoroughly and not rely on the teaching staff to provide test cases.

Here's a scenario to test your p2p network construction code using four hosts. At the first host, start your peer code with max peers set to 2. Next start a second peer with max peer set to 3, connect it to the first peer. Then start a third peer with max peer set to 2 and connect it to the second peer. If your automatic join code is working, peer 3 should then also join peer 1. Finally, start peer 4 with max peer set to 1 and try to connect it to peer 3. Peer 4 should fail to connect to peers 3 and 1 but successfully connect to peer

2.

To test your search code, search for an image that is at least 2 hops away. Search for a non-existing image, and search for an image that is held by more than one peers.

Support Code

The support code for Labs 1 and 2 form the support code of this assignment. So that you don't feel like you're only filling in functions and not having any chance to write your own program from scratch, we are not providing further support code for this assignment. If you have not been able to complete Lab 1 and would like the solution so that you can complete this assignment, you may choose to forfeit the 20 points associated with it and obtain a solution from us. Similarly for Lab 2, and tasks 2, 3, and 5 of this assignment. Sharing the code and solutions will be considered cheating and will be reported to the Honor Council.

Remember not to use any libraries or compiler options not already used in the Makefile to ensure that we will be able to build your code for grading. If we can't compile your code, you will get 0 point.

Submission Instructions

Test your compilation! Your submission must compile **without** errors. Code that does not compile will be heavily penalized.

Create a writeup in **text format** that discusses:

1. Your platform and its version - Linux, Mac OS X, or Windows, and which version and flavor of each.
2. Anything about your implementation that is noteworthy.
3. Feedback on the assignment.
4. Name the file writeup-*uniqname*.txt.
   For example, the person with uniqname *tarukmakto* would create *writeup-tarukmakto.txt*.

Your *PA1 files* comprises your `writeup-uniqname.txt` and your source code files.

To turn in your PA1:

1. Email your IA/GSI the SHA1's of your *PA1 files*. Use "EECS489: PA1 Submission" as your email's "Subject:" line. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *PA1 files* by pointing your web browser to Course folder and navigate to your `pa1` folder under your uniqname. Or you can `scp` the files to your `lab1` folder on IFS:
   `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/pa1/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 email will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. Try not to email your SHA1 to your IA/GSI until you've

finalized your code. You don't want to annoy them.

Do NOT turn in an archival (`.zip` or `.tgz`) file, instead please turn in your solution files individually. Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object files, executables, or images) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s and any other logging statements you've added for debugging purposes. You should debug using a debugger, not with `printf()`'s or equivalent. If we can't understand the output of your code, you will get zero point.

General

It is part of the Honor Code of this course that the overall design and final details and implementation of your programming assignments must be your own. If you're stuck in either the design, implementation, or debugging of the assignment, you're allowed and encouraged to consult with your classmates. However, the original design and final implementation details must all be your own. So you cannot come up with the original design *together* with your classmates. You can consult your classmates *only after* you've come up with your own design but ran into some specific problems. Similarly for the implementation, you cannot consult your classmates prior to writing your own implementation. And in all cases, you're not allowed to look at any of your classmates' source code, not even in order to help them to debug. The same applies to design and implementation from previous terms.

Coding style

Use a reasonable organization for your overall program:

> Design a fairly reasonable class structure. On the one hand, don't stick everything into one class/struct. On the other hand, don't be bureaucratic and require the reader to follow one class definition after another to find a single line of code wrapped in n layers of methods, with each method doing nothing but calling the next one. If the way you design your code feels sloppy to you, it probably is. Utilize multiple files in a way that is consistent with the general use of C/C++. Don't use more files than necessary, you don't have to put each class/struct in a separate file of its own.

Don't use literals!

> Use either `const`, `enum`, or `#define` to give your literals meaningful names. We do deduct points for *each occurrence of literals*, even if it is the same one. The only exceptions will be for loop counter, command-line options, NULL(0) and TRUE/FALSE(1/0) testing/setting, and help and error messages printed out to user.

Use reasonable comments:

> Explain what each class does and what each data member is used for. A one or two line description of most member functions is also desirable. Where you use non-standard coding techniques, document them. **List your name and the date last modified for each file.**
>
> Remember that a useless comment is worse than no comment at all.
> `int temp; // declare temp. variable`
> would be an example of a useless comment which just makes code harder to read!

Use reasonable formatting:

> From indentation alone, it should be obvious where a given code block ends. Avoid lines that wrap

in an 80 column display wherever possible. Your code should be tight, compact, and visually tidy. Don't let bits and pieces fly off every which way. Your code is not abstract painting.

Variable names:
: Use reasonable and informative variable names, but limit name size to a reasonable length. A 40-character name better has a very good reason to exist. Variable names like 'i' and 'j' can be reasonable, but you should not use such variables to store meaningful long-term data. Other than LCV (loop control variables) you should use descriptive names for your variables, functions, classes, methods, structures, etc.

Reduce, Reuse, and Recycle your code, algorithms, and structures:
: Try using inheritance, templating, polymorphism (virtual function), or similar methods to reduce the size of your code. Do not unnecessarily duplicate code. Less code leads to less debugging. If you find yourself rewriting basically the same code more than once, stop and try to see if you can somehow reuse the code by making it a function call or implementing a polymorphic function.

Unreadable code can cost you up to 10 points!

Empirical efficiency

We will check for empirical efficiency both by measuring the memory usage and running time of your code and by reading the code. We will focus on whether you use unnecessary temporary variables, whether you copy data when a simply reference to it will do, whether you use an O(n) algorithm or an O(n^2) algorithm, but **not** whether you use `printf`'s or `fprintf`'s. Nor whether your ADTs have the cleanest interfaces. In general, if the tradeoff is between illegible and fast code vs. pleasant to read code that is unnoticeably less efficient, we will prefer the latter. (Of course pleasant to read code that is also efficient would be best.) However, take heed what you put in your code. You should be able to account for every class, method, function, statement, down to every character you put in your code. Why is it there? Is it necessary to be there? Can you do without? Perfection is reached not when there is nothing more to add, but when there is nothing more that can be taken away, someone once said. Okay, that may be a bit extreme, but do try to mind how you express yourself in code.

Hints and advice

- Design your data structures and work through algorithms on paper first. Draw pictures. Consider different possibilities *before* you start coding. If you're having problems at the design stage, come to office hours. After you have done some design and have a general understanding of the assignment, re-read this document. Consult it often during your assignment's development to ensure that all of your code is in compliance with the specification.
- Always think through your data structures and algorithms before you code them. It is important that you use efficient algorithms in this programming assignment and in this course, and coding before thinking often results in inefficient algorithms.
- Make sure you don't clutter `stdout` with unnecessary output. Use `gdb` to debug.
- You shouldn't print to `stderr` unless there is an error.
- Systems programs has a lot of cases to consider and even the simplest program can sometimes be tedious to code. This is not a short programming assignment. Start it immediately.
- The teaching staff will be happy to help you track down bugs, but you have to fix them yourself once they are found. We will not help you track down a bug unless you can show us *in gdb* where you suspect the bug to be. That is, you need to show us that you have tried your best to track down the bug, and that you have used gdb.

To encourage early start on the assignment **we will stop helping you to debug 48 hours before the due date**. For a Monday midnight deadline we stop helping you at midnight on the Saturday prior.

If any part of this document is unclear, ambiguous, contradictory, or just plain wrong, please let one of the teaching staff know. Have fun coding!