416 Distributed Systems: Project 1 [BlockArt]

Due: Feb 16 at 11:59PM

Winter 2018

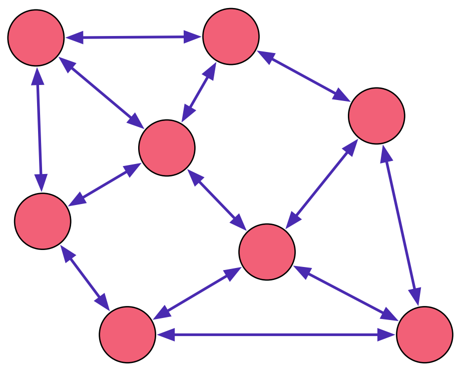
In this project you will develop **BlockArt**, a blockchain-based system to support collaborative computer art projects. Each block in the chain will add/remove visual elements on a canvas that represents the artwork. Your blockchain network must be built to support thousands of nodes and must use a flooding protocol to disseminate operations on visual elements and generated blocks. Applications can interact with your blockchain and participate in an art project instance by using an API that you will provide. Your blockchain will validate authenticity of operations (verify that a particular application introduced an element) and your system must use a proof of work algorithm to guarantee fairness (artist applications with more computational power get to contribute more elements to the canvas).

There are several extra credit options, including adding new shapes to the system, and developing a browser viewer that will allow a user to view the art work and watch it evolve over time.

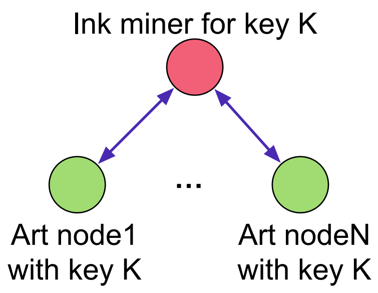
You will deploy and test your BlockArt system on the Azure cloud.

System overview

There are three types of nodes in the system: some number of *ink miners*, some number of *art nodes*, and a single *server* node. There is one server per art project/canvas, and an ink miner/art node can only participate in one canvas at a time. The server provides a simple interface to register/retrieve ink miners and keeps track of ink miner failures. The ink miners form a peer-to-peer graph (based on information provided by the server). Each miner is connected to some number of other miners (at least three in the diagram below):


Ink miners network

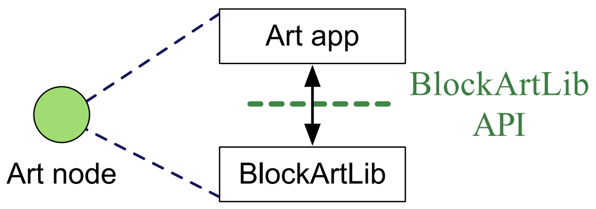
Ink miners. The ink miner is initialized with information about the server's address and a public/private key pair. The miner should connect to the server to register, retrieve settings, and lookup other ink miners to connect to. The miner can *only* communicate with miners that it learns about from the server. If the number of miners it knows about dips below a threshold specified in settings, `MinNumMinerConnections`, (because of failures), then the miner should ask the server for more miners (but not otherwise). At some later point, the miner may receive a connection from one or more art nodes. The miner must be able to support several simultaneous art nodes. The miner must validate that the art nodes connecting to it know the public/private keys that the ink miner is configured with.



An ink miner has two roles: it mines ink on behalf of a specific private/public key pair, and it also participates in the miner network. Miners have the following responsibilities:

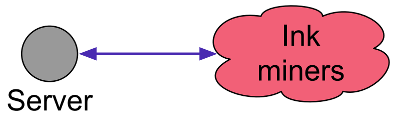
- The ink miner must disseminate operations sent to it by art nodes that are connected to it to the rest of the ink miner network. The miner must also help other miners with their operation dissemination.
- A miner mines *ink* for the applications associated with the art nodes to use. Without sufficient ink, the application cannot add new elements to the canvas. Mining is done using a proof of work algorithm based on what you developed in assignment 1.
- By solving the proof of work, the miner generates a block for the blockchain and some amount of ink that is credited to the public/private key pair that it was initialized with. The generated block includes the set of operations that the miner knows about (or is a no-op block; see below).
- A miner validates mined blocks it receives and disseminates valid blocks (that it mined locally and that other miners mined) to the rest of the miner network.
- Periodically, the miner must ping the server with a heartbeat.

Art node. An art node provides an interface to a local application (art app) through a Go library called *blockartlib* to add/remove elements on the canvas:



The art app will initialize blockartlib with information about the miner's address and a public/private key pair. The art node will connect to miner and validate that this miner mines on behalf of the right key pair. The art node can *only* communicate with the one ink miner that was specified on the command line. The application will then use the local blockartlib instance to perform operations on the global canvas and to retrieve information about the blockchain. Art nodes do not communicate between one another.

Server. The server helps miners find each other. We will give you a simple version of the server. You can change the server code in your development, but you cannot change the interface between the server and the miners.



APIs. The BlockArt system has several APIs. It is important to understand which ones are your responsibility to design and implement. Here is a summary:

- art app — blockartlib : API specified below; you will have to implement this API.
- ink miner — server: API specified below, the server side of the API is implemented and given to you.
- art node — ink miner: not specified, and up to you to design and implement.
- ink miner — ink miner: not specified, and up to you to design and implement.

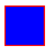
You have substantial flexibility in the design and implementation of this project. Keep things simple and design as much of the system as you can before starting on the implementation.

BlockArtLib API

This library exposes an interface that is detailed in the blockartlib.go file. You are **not** allowed to change this interface. The library generally has two kinds of calls: calls to manipulate the canvas, and calls to retrieve information about the blockchain. All of the calls require connectivity and return `DisconnectedError` if the art node is no longer connected to the network. All of the calls are blocking calls: they do not return until they have finished successfully, or an error occurred.

- `canvas, settings, err` ← OpenCanvas(minerAddress, private/public Keys)
  - Opens a new canvas, providing the miner address that is mining on behalf of the public and private key pair arguments. Returns error, or if error is not set, then a canvas reference/instance and settings for this canvas instance. Error can be `DisconnectedError`.
- `inkRemaining, err` ← canvas.CloseCanvas()
  - Closes the canvas/connection to the BlockArt network. Returns error, or if error is not set, then the amount of remaining ink owned by this application instance. Error can be `DisconnectedError`.
- `shapeHash, blockHash, inkRemaining, err` ← canvas.AddShape(validateNum, shapeType, shapeSvgString, fill, stroke)
  - Adds a new shape of type shapeType and description string shapeSvgString to the canvas. `validateNum` specifies the number of blocks (no-op or op) that must follow the block with this operation in the block-chain along the longest path before the operation can return successfully. Returns error, or if error is not set, then returns the hash of the shape, the blockHash of the block to which this shape was added, and ink remaining. Error can be `DisconnectedError`, `InsufficientInkError`, `InvalidShapeSvgStringError`, `ShapeSvgStringTooLongError`, `ShapeOverlapError`, `OutOfBoundsError`.
  - Your library must provide support for the `PATH shapeType`. As extra credit you may add other shape varieties. The path shape is

modeled after the SVG path. You only have to support the following path commands: `M, m` (move to), `L, l` (line to), `H, h` (horizontal line), `V, v` (vertical line), `Z, z` (draw straight line back to starting position). Capital letters denote commands relative to absolute canvas coordinates, while the lower-case letters denote commands against relative coordinates. See this guide for more information.

  - The `shapeSvgString` captures the command sequence. This string can be at most 128 characters long. For the `PATH` shape type, this string must be a legal svg `d` attribute to the svg `path` element and can include only the commands listed above. If this string is too long or invalid in format, then a `ShapeSvgStringTooLongError` or `InvalidShapeSvgStringError` should be returned, respectively.
  - The `fill` argument is a string that determines whether the shape has no fill: `transparent`, or the shape has fill of a certain color, e.g., `red`. A shape with transparent fill uses no ink for the area inside the shape; a shape with non-transparent fill also uses ink for the filled area. For example, the `PATH` command "M 0 10 H 20" would use 20 units of ink: But, the command "M 0 0 H 20 V 20 h -20 Z" with non-transparent (blue) fill and non-transparent (red) stroke would use 20*20 (area) + 20*4 (length) = 480 units of ink:  Here are the precise calculations you must implement:
    - transparent fill + non-transparent stroke ink used = length
    - non-transparent fill + transparent stroke ink used = area
    - non-transparent fill + non-transparent stroke ink used = area+length
    - (transparent fill + transparent stroke: `InvalidShapeSvgStringError`)
  - An `InsufficientInkError` should be returned if the shape requires more ink than what the application instance has currently (i.e., ink mined by the miner, minus ink used to create shapes by applications with the corresponding key pair in the past). Ink mined to generate the block that will contain this shape operation does *not* count.
  - `OutOfBoundsError` is returned if the shape exceeds the bounds of the canvas. `ShapeOverlapError` error is returned if this shape intersects/overlaps (in at least 1 point) with a shape that was added previously by another application, but not this application.
  - You do not need to support arbitrary `shapeSvgString` strings (though you can, if you want to). Here are some constraints to make the parsing easier. Expect that `shapeSvgString` abides by the following:
    - Uses only spaces for delimiters (no commas)
    - Always has a space between the command name and the arguments (e.g., "M 1 2", not "M1 2")
    - Each command in the string has a fixed set of arguments (e.g., no "M 20 20 l 50 20 15 25", 2 extra args to l)
    - Always starts with the "M" command (necessary since the set of operations in the block is unordered)
    - There is one "M" command (at the start of the string) if fill is non-transparent
  - Stroke and fill are required arguments. Stroke and fill args cannot both be empty strings and cannot both be set to "transparent". In these cases return an `InvalidShapeSvgStringError`.
- `svgString, err` ← canvas.GetSvgString(shapeHash)

  - Returns error, or if error is not set, then returns the encoding of the shape identified by shapeHash as an svg string. For example a PATH shape with command "M 0 0 L 20 20" and stroke "red" and fill "transparent" would return the following string:
    `<path d="M 0 0 L 20 20" stroke="red" fill="transparent"/>`
    Which can be rendered in a browser (on a 20x20 canvas) to look like:
    Can return `DisconnectedError` and `InvalidShapeHashError`. An `InvalidShapeHashError` should be returned if the shape with shapeHash cannot be found.
- `inkRemaining, err` ← canvas.GetInk()
  - Returns the amount of ink currently available to the application. Can return `DisconnectedError`.
- `inkRemaining, err` ← canvas.DeleteShape(validateNum, shapeHash)
  - Removes a shape with shapeHash from the canvas. `validateNum` specifies the number of blocks (no-op or op) that must follow the block with this operation in the block-chain along the longest path before the operation can return successfully. Returns the amount of ink currently available to the application. Deleting a shape credits the application with the ink that the shape was using (e.g., if a PATH shape "M 0 10 H 20" is deleted, then the application's ink supply would go up by 20 units). Can return `DisconnectedError` and `ShapeOwnerError` errors.
  - `ShapeOwnerError` is returned if this application did not create the shape with shapeHash (or if no shape exists with shapeHash).
- `shapeHashes, err` ← canvas.GetShapes(blockHash)
  - Returns the set of all shape hashes that belong to a block identified by the hash blockHash. Can return `DisconnectedError` and `InvalidBlockHashError` if no block with blockHash exists.
- `blockHash, err` ← canvas.GetGenesisBlock()
  - Returns the block hash of the genesis block in the blockchain. Can return `DisconnectedError`.
- `blockHashes, err` ← canvas.GetChildren(blockHash)
  - Returns the set of all children blocks for block identified by the hash blockHash. Can return `DisconnectedError` and `InvalidBlockHashError` if no block with blockHash exists.

Consult the sample art application that uses this interface for an example of how this interface can be used.

Notes on the BlockArtLib API.

- If the fill is set to transparent, then the application should be able to construct an arbitrary path using the allowed commands. For example, the shape might self-intersect at one or more points.
- If the fill is not transparent, then the application is constrained to constructing one **simple closed curve** (no self-intersections, no end points, and completely encloses an area), such that the first and last points along the path are the same. This last constraint could be satisfied by using a `Z` command, or by explicitly specifying identical x,y coordinates (in the absolute or relative coordinate systems).
- In the API `inkRemaining` is a uint32. This means that you have to do some rounding in case of fractional ink. You should always **round up** the value returned for `inkRemaining`.

- Shapes generated by different art applications cannot overlap/intersect. However, shapes added by the same application *can* intersect. Note that the overlap region does **not** save ink for the application (e.g., repainting the same line segment uses up ink).
- The shape spec is a translation of the svg spec. When in doubt about what the shaded area looks like, or if a certain command sequence is legal, check with the svg spec, run an experiment in a browser, or find/use a tool.

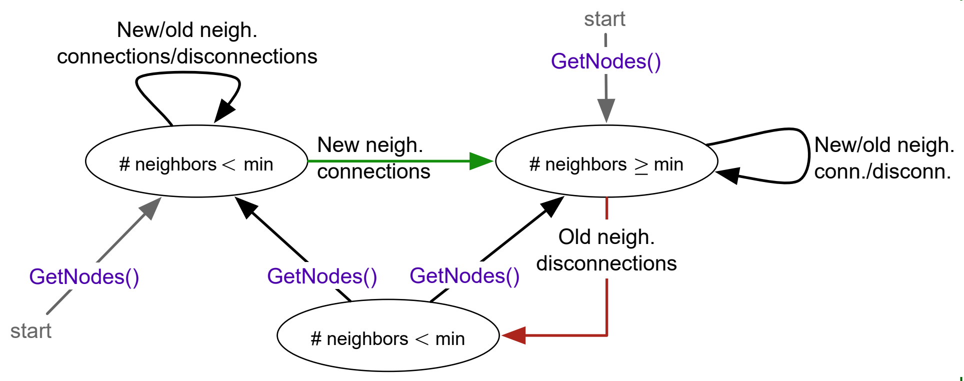
Server API

The server listens to connections from ink miners on a known TCP IP:port.

Server — ink miner API. An ink miner can invoke three kinds of TCP RPCs against the server:

- `settings, err` ← Register(address, publicKey)
  - Registers a new miner with an address for other miner to use to connect to it (returned in `GetNodes` call below), and a public-key for this miner. Returns error, or if error is not set, then setting for this canvas instance. Returns `AddressAlreadyRegisteredError` if the server has already registered this address. Returns `KeyAlreadyRegisteredError` if the server already has a registration record for publicKey.
- `addrSet,err` ← GetNodes(publicKey)
  - Returns addresses for a subset of miners in the system. Returns `UnknownKeyError` if the server does not know a miner with this publicKey.
- `err` ← HeartBeat(publicKey)
  - The server also listens for heartbeats from known miners. A miner must send a heartbeat to the server every `HeartBeat` milliseconds (specified in settings from server) after calling Register, otherwise the server will stop returning this miner's address/key to other miners. Returns `UnknownKeyError` if the server does not know a miner with this publicKey.

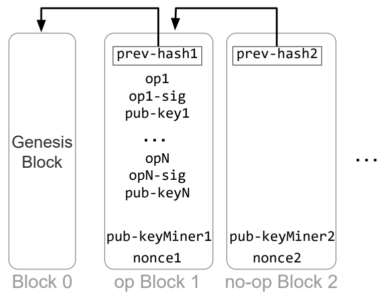
The ink miner should follow the following state machine in calling `GetNodes` ("min" in the diagram stands for `MinNumMinerConnections`):



Blockchain details

Block generation. Each miner must implement a *mining* procedure by which it can generate *ink* and add a new block in the block chain. A node can only compute one block at a time and *cannot* work on multiple blocks simultaneously. There are two kinds of blocks: *no-op* blocks and *op* blocks.

- **A no-op block** does not contain any operations. These blocks are generated so as to prevent pre-computation of long block sequences by malicious nodes and to validate existing operations (the validateNum argument to AddShape and DeleteShape controls validation from the art application's side). All nodes should always be working on no-op blocks in the background, constantly generating these and adding them to the block-chain.
- **An op block** contains several operations. When a client notifies blockartlib of an operation, the local art node sends the operation to the miner. The miner immediately stops working on no-op blocks (but not op blocks) and switches to work on an op block to integrate the new operation into the block-chain. The miner can and should integrate multiple operations that it knows about (potentially broadcast/generated by other miners) into one op block.



Block chain. Miners maintain a tree representation of the block chain. The *chain* is the longest path in this tree, starting at the *genesis* block whose hash is specified by the server. A miner should only compute no-op and op blocks along the chain, and not along any shorter path in the tree. In the case that there are several (longest) chains, the miner should (1) pick the one that does not cause a validation error for the current op block it is generating, or if no op block is being generated or none cause a validation error, then (2) the miner should pick among the chains uniformly at random.

Block data structure. An op block is a data structure that contains *at least* the following data:

- A hash of the previous block in the chain (prev-hash)
- An unordered set of operation records; each operation record should include:
  - An application shape operation (op)
  - A signature of the operation (op-sig)
  - A public key of the art node that generated the op (used to validate op/op-sig)
- The public key of the miner that computed this block (pub-keyMiner)
- A 32-bit unsigned integer nonce (nonce)

Block hashes (e.g., prev-hash) must be an MD5 hash and must contain a specific number of zeroes at the *end* of the hash (a constant specified during registration with the server). A op block's hash is a hash of [prev-hash, op, op-signature, pub-key, nonce]. The goal of the proof-of-work algorithm is to find a nonce such that the block's hash contains the required number of zeroes. The more zeroes, the longer it takes to generate the block (find a nonce that works).

Note that the `shapeHash` argument in the blockartlib API can be the signature of the shape (op-sig). However, it does not have to be the signature; it can be any string that **uniquely** and **consistently** identifies the shape globally in the blockchain. Consistently here means across different art apps and different API call instances.

Note that a block hash computed for an op block or a no-op block by a miner A will always be different from blocks generated by other miners (for the same op/no-op) and will always differ from other blocks generated by miner A. This is because a block contains a prev-hash, which uniquely identifies its position in the blockchain tree, and a block contains a public key (e.g., known by node A), which makes each block unique to A.

A no-op block is identical to an op block except that it does not include an op. Its hash is similarly computed using a proof-of-work algorithm.

Block validation. This project does not assume trust between miners. For example, miners may attempt to cheat (spend ink they don't have) and steal (spend ink belonging to other nodes). That is, you should design your miner with the assumption that other miners may be malicious. A key aspect of your design must be a set of routines to validate and check each piece of information that you receive from other miners. It is especially important to validate blocks, since un-validated blocks can completely undermine your system.

Here is a minimal set of validations each miner should perform:

- Block validations:
    - Check that the nonce for the block is valid: PoW is correct and has the right difficulty.
    - Check that each operation in the block has a valid signature (this signature should be generated using the private key and the operation).
    - Check that the previous block hash points to a legal, previously generated, block.
- Operation validations:
    - Check that each operation has sufficient ink associated with the public key that generated the operation.
    - Check that each operation does not violate the shape intersection policy described above.
    - Check that the operation with an identical signature has not been previously added to the longest chain in the blockchain. This prevents operation replay attacks.
    - Check that an operation that deletes a shape refers to a shape that exists and which has not been previously deleted.

BlockArtLib API. Each art node must implement the blockart lib interface. For this, the node must (1) store all operations that have acted on the canvas thus far, and (2) it must respond to drawing operations against this version of the canvas. The library must continue to respond to application operations even though the block-chain underneath changes.

Other notes:

- Two concurrent and conflicting shapes (i.e., intersecting shapes added by two applications with different keys) cannot be added to the same block. Your design must guarantee this.
- Conflicting shapes also cannot appear along the longest path in the blockchain (this check must be part of validating an operation). This means that if two applications with different keys simultaneously issue operations for shapes that intersect, then one of the applications should (eventually) get back a `ShapeOverlapError`. The other operation must succeed. In general, exactly one of the conflicting operations **must** succeed, while the others must get a `ShapeOverlapError`.

Handling failures

Ink miners and art nodes may fail stop. If an art node fails mid-operation, the miner may abort or it may commit the art node's operation. If the ink miner that an art node is connected to fails, the blockartlib at the art node should return a `DisconnectedError` to the art app.

Ink miners and art nodes should not store/cache any state on disk. A failure therefore completely wipes out the node's state.

As long as an ink miner is connected to at least one other ink miner, it should continue to operate in the miner network. The ink miner should only call the server's `GetNodes` to retrieve more mining nodes if the number of non-failed miners that it is connected to drops below `MinNumMinerConnections` (returned as part of settings from the server).

Azure deployment

The Azure cloud is composed of several data-centers, located world-wide. Each data center hosts many thousands of machines that can be used (i.e., rented) for a price. In this project you may use the Azure cloud to deploy and test your solution in the wide area.

Although you will test and deploy your system on Azure, it will have nothing Azure-specific about it, e.g., it should be possible to run your system on the CS ugrad servers without any code modifications.

Using Azure: stop VMs when not using

We prepared a google slides presentation covering the basic workflow of getting a VM running on Azure for this/future assignments. To setup the Go environment in a VM you can use the azureinstall.sh script.

The default Azure subscription comes with a limitation of 20 cores per region. For this assignment you should not need cores above this limit.

Use this site to check your account balance.

Access information posted to piazza.

A key detail is that each second that your VM is running it is draining your balance (yikes!). You should **STOP your VMs when you are not using them.** It's up to you to police your own account.

Assumptions you can make

- The server is always online, does not fail, and does not misbehave.
- The public key uniquely identifies an ink miner and an art node.
- No network failures (e.g., partitions or failed links).
- A timeout of 2s can be used to detect art node and ink miner failures.
- The server will send an identical/consistent settings to all miners and art nodes.
- A node does not fail until after the `OpenCanvas` call has returned.
- A maximum of 65,535 ink miners will be in the network.

- The miner graph will remain connected: miner failures will not induce a partition of the miner graph.
- The server will return a minimum of 0 and a maximum of 255 ink miners in response to a registration.
- Neither ink miners nor art nodes are behind a NAT or Firewall that makes them unreachable to each other or to the server.
- Miners will follow the protocols you design and will not deny service to other miners (e.g., refuse to communicate with some miners).

Assumptions you cannot make

- There is a known ordering on when nodes fail or join the network.
- There is a known number of nodes that will join the system or that will fail.
- An ink miner or art node that has failed will later re-join the system.
- Miners, art nodes, and the server have synchronized clocks (e.g., running on the same physical host).
- Reliable network (e.g., if you use UDP for communication, then expect loss and reordering)
- You are running on a specific type or version of an OS.

Implementation requirements

- The client code must be runnable on Azure Ubuntu machines configured with Go 1.9.2 (see the linked azureinstall.sh script and the Google slides presentation for more info).
- You must use the crypto/ecdsa Go library for your public-private keys, op signing/validation, etc.
- The `ink-miner.go` file should live at the top level of your repository in the *main* package.
- The blockartlib implementation should live inside the `blockartlib/` directory at the top level of your repository (it should be possible to run `go run art-app.go` at the top level).
- You must support the blockartlib API.
- Your solution can only use standard library Go packages.
- Your solution code must be Gofmt'd using gofmt.

Solution spec

(1) Write a go program called `ink-miner.go`, (2) a Go library called `blockartlib.go` that behave according to the description above, and (3) an application that uses blockartlib and produces an html file as output that contains an svg canvas that is the result of the application *distributed* activity (the application must be runnable at different ink miner node instances simultaneously).

Ink miner's command line usage:

```
go run ink-miner.go [server ip:port] [pubKey]
[privKey]
```

- [server ip:port] : the IP:port address of the server in the BlockArt network.
- [pubKey] [privKey]: the public and private key pair (represented as two string command line arguments) that this miner should use to validate connecting art nodes and towards which it should credit mined ink.

Starter code

You can use a stub implementation of the blockartlib API and a client application that uses it as starters for your system:

- blockartlib.go (blockartlib)
- art-app.go (art application)
- proj1-server.zip (server.go, config.json, tester.go)

Grading scheme

Note that this is a draft that will remain a draft until next week Monday evening after we have attempted this rubric with the first set of teams. It captures my ideal version of what I want in the demo. But, teams might run out of time, it might be impractical to modify code live, many VMs might prove too complex, etc. So, this rubric/demo sequence might get simpler based on our experience with demos on Monday, Feb 19th.

At the high level your mark for project 1 is 20% of your final mark and has these components:

- Code: 10%
- Demo: 10%
- Peer review multiplier

Note that the demo actually exercises both the Code and the Demo portion of your mark. So, the best way of thinking about the demo rubric below is that it is 100% of your mark for project 1. Of course, we reserve the right to look at your code on our own if we want to to further convince ourselves about some functional aspect or check that you do indeed implement some particular piece (e.g., flooding of blocks).

The peer review multiplier will be an online survey. In the survey you will be asked to partition some number, like 100, into as many parts as there are team members in your project 1 team. The survey results will only be available to instructors (not revealed to your team members). The rough formula I will use is that the average of fractions that your team members assign you relative to an equal split will become a multiplier for your demo mark. In some cases, if there are substantial discrepancies (e.g., teammates give each other 0s), I might interview team members to get to the bottom of the story and figure out the multiplier accordingly.

The demo has four parts, along with percentage of your project mark:

- Part A (40%) : you demo for us your system using your own code (your own server/miner/art apps). This is also your chance to show us any extra credit options that you have implemented.
- Part B (30%) : you demo for us your system with our server (+ your miners/art apps)
- ~~Part C (10%) : you demo for us your system with our server and changes to some of your art apps (+ your miners)~~
- Part D (**30%**) : we ask you design-level and code-level questions about your system

Except for the first 4 demos, each demo will have 2 instructors -- either two TAs or Ivan and one of the TAs. One of us will be taking notes, and the second person will be laser focused on your demo -- what you are doing, how you are doing it, clarifying what's going on, etc.

Each team has a guaranteed slot of 20 minutes. Each part of the demo is expected to take about 5 minutes. Note that 20 minutes may be a hard cut off, especially in cases where there is another team scheduled after your team. (We give ourselves about 5-10 minutes to deliberate your mark and review the demo after you leave the room. The more time we have for this, the more likely that you will get a fair shake).

Note that your demo will be run in our environment. That is, when you enter the demo room, we will have a set of Azure VMs up and running and ready for your demo. Each VM will have:

- Go version 1.9.2 installed
- Emacs/vi/vim installed
- Your code cloned/checked out from stash (last version before deadline)
- A file at top level of your repo called server.ip containing one line with the IP of the VM where the server should run
- A file at top level of your repo called miners.ip containing as many lines as there are VMs available, with each line containing an IP of a VM where a miner should run

Your demo in part (A) must obey the following parameters:

- The server must run on a separate VM (specified in server.ip)
- Each miner must run on a separate VM (specified in miners.ip)
- Art-apps must be co-located with the miner that they are connected with (run on same VM as the miner that they are connected to)
- There must at least 1 miner with more than 2 connected art apps
- There must be exactly 1 miner without any art apps (others have at least 1 art app)
- Your miners network must include 5-10 miners
- You must use PoW difficulty less than or equal to 4 (for both op and no-op blocks)
- Your part A must run and output the HTML file that we you can download/preview in under 5 minutes. We will try not to cut you off, but if you go over you will have less time for the the other parts of the demo. You must track your own time and be fast.
- You kill/stop all of the art apps and miners once you are done with Part A

Part B (with our server):

- We reveal a server that is conveniently running on Azure as a VM at a certain IP:port.
- This server respects the server API but it constructs a miner topology that is unknown to you (a focus of this part of the demo is to see if your code works with an arbitrary topology).
- You start all of the miners to connect to this new server
- You start the art apps and run through your demo as in Part A
- You kill/stop all of the art apps and miners once you are done with Part A
- We consider the HTML file output (it should be identical modulo race conditions; which you can explain to us as part of the demo).

~~Part C (with our server + changes to some of the art apps):~~

- ~~We give us the file names for two art apps that are connected to two different miners in your network~~
- ~~We hack the art apps -- delete code, and add some code. For example,~~

    - ~~We might call AddShape on a square that is HUGE (should fail with insufficient ink error)~~
    - ~~We might call AddShape on two lines that intersect (should fail with overlap error)~~
    - ~~We might add a loop that counts number of blocks in the chain, outputs that number, sleeps, and re-loops.~~
- ~~You start all of the miners~~
- ~~You start all of the art apps~~
- ~~We observe the modified art apps to see if they behave in the expected manner~~

Part D (design Q/A):

- We will ask each person on the team one question from a list of questions that we have curated for this part. How many people we ask will depend on time.
- We will try to make sure there is a whiteboard available for you to draw diagrams if you need one.
- Each person on the team should attempt to answer the question on their own first. If the person is unable to do this, then the rest of the team (whoever else wants to help) can jump in and help answer the question.
- The questions are high-level design questions about your system. They are the kinds of questions that everyone on the team should know the answer (at least to some degree!). Some examples:
    - What is your protocol for distributed operations in the miner network?
    - What is your approach for determining shape intersections?
    - How does your system figure out ink remaining to return to the art app after it calls AddShape?
    - What happens if the miner fails before it is able to disseminate an operation to other miners?
    - How does miner A check the signature of an operation generated by art app connected to miner B?
- These questions may escalate into discussion about your design choices more broadly. (So, we may not get to ask everyone on the team a question.)

Nice to have, or strongly recommended for the demo:

- Strongly recommend to pre-generate a set of ECDSA keys for your demo use (pre-generate a bunch, like 20). Commit these keys into your repo (note: this is bad practice in general; but ok for course project demos). You can commit them in whatever format you want -- as 1 file with a bunch of keys, or 1 file per key, whatever you think would best facilitate their usage during the demo.
- Strongly recommend including (comprehensible) stdout/logging in your codebase so that we can track things as they occur on the terminal. If you lack this, then we can only go by the output html string, and if that doesn't work, then it will be hard (impossible) for us to justify any partial marks.

What to bring for the demo:

- You'll need your own laptop to drive the demo. I recommend bringing several so that others can monitor the system, and you have some redundancy + 8 hands can type faster than 2 hands.
- Team spirit.

Extra credit

This project is extensible with three extra credits:

- EC1 [1% of final mark]: Add the circle shape to blockartlib, modeling it after the svg circle element. Add the appropriate ShapeType, encode the required circle attributes in `shapeSvgString`, and include support for fill and stroke attributes. Get ready to do some math to figure out pair-wise shape intersections, circle perimeters/areas, canvas out of bounds errors, etc.
- EC2 [1% of final mark]: Create an art application that includes a built-in web server that serves a page that can be accessed in a browser. The page should display a navigable and visual history of the block-by-block evolution of the svg canvas in the BlockArt network. The browser-side JS can use whatever libraries you want.
- EC3 [0.5% of final mark]: Similar to EC2, but allow the user to add shapes to the canvas in real-time and observe the resulting svg canvas (as stored within the blockchain)

Make sure to follow the course collaboration policy and refer to the assignments instructions that detail how to submit your solution.

Last updated: February 20, 2018