Review: Comm. as shared state

• Communication is less than most think
  – Just syntax – not semantics or intent

• Information is based on states
  – Which is based on entropy (disorder)

• We can model how state evolves
  – Each side models the other
  – Successive steps in models are how we go from sharing state to transferring files

• Noise decreases the information we can pass
  – Encodings can correct errors
  – But cannot break the Shannon limit
Communication: Roadmap

• The imperfect channel
• Making the channel real
• Automating the channel
What is “encoding a block”?

• Forward error correction
  – A way to detect and correct errors without asking for more information

• Examples:
  – Parity
  – Majority
  – Hamming
  – Reed-Solomon
Parity

• Sender
  – Add one bit to ensure a block has an EVEN (or ODD) number of 1’s
  – 01011 -> 010111

• Receiver
  – Check to see if the pattern has the correct number of 1’s
  – 010111 is OK
  – 011111 is BAD
What can parity help with?

• Error detection
  – Detects any ODD number of bit errors in the block
• Cost:
  – One extra bit per block
• Limits:
  – Won’t detect any EVEN number of errors (regardless of parity type)
  – Cannot correct the errors
Majority

- Sender uses a repetition code
  - E.g., send each bit three times
  - 01011 -> 0011100011111

- Receiver uses majority voting
  - For each triplet, pick the majority value
  - 0011101111011 becomes 01111
What can majority help with

• **Error detection**
  – Detected when a group is not all the same
  – Can locate errors to each group they occur

• **Error correction**
  – YES, for $k < \frac{N}{2}$ errors per group

• **Cost:**
  – $N$ times longer

• **Limits:**
  – Very high cost
Hamming

• Combines parity with sets

• Sender
  – Use the algorithm to generate parity codes within various subsets of the bits

• Receiver
  – Use the algorithm to check parity codes within various subsets. When a parity check fails, it indicates the bit position of the error
A Hamming Code Example

• Let’s say we have a 15 bit data item

\[d_1 \ d_2 \ d_3 \ d_4 \ d_5 \ d_6 \ d_7 \ d_8 \ d_9 \ d_{10} \ d_{11} \ d_{12} \ d_{13} \ d_{14} \ d_{15}\]

• We want to send it on a noisy channel and be able to correct a 1-bit error

• Add 5 parity bits
  – Why 5? We’ll see in a minute

• But don’t use them as a single 5-bit number

• Have each parity bit cover a subset of the overall bits
Building the Hamming Code

• We’re going to send 15 bits encoded as 20 bits
  – Adding 5 parity bits to the 15 data bits
• Where do we put the 5 parity bits?
• Not at the end
• Scattered through the 20 bit encoded value
• OK, so which bits are parity bits?
• Any bit whose binary value for position contains only one 1
  – Bit 1 (1), bit 2 (10), bit 4 (100), bit 8 (1000), bit 16 (10000)
Adding the parity bits

<table>
<thead>
<tr>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
<th>d8</th>
<th>d9</th>
<th>d10</th>
<th>d11</th>
<th>d12</th>
<th>d13</th>
<th>d14</th>
<th>d15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p4</th>
<th>p8</th>
<th>p16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

| 1  | 10  | 11  | 100 | 101  | 110  | 111  | 1000 | 1001  | 1010  | 1011  | 1100  | 1110  | 1111  | 10000 | 10001 | 10010 | 10011 | 10100 |
|----|-----|-----|-----|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
|     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|     |     |     |     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |

What does each parity bit cover?
What do we mean by “cover”?

- The parity bit is calculated in the usual way
- But considering only the bits it covers
- So each of the five parity bits is computed differently
  - Considering a different (but overlapping) set of data bits
What does this buy us?

• If one bit is flipped, some parity bits will come out wrong, when checked
• Which indicates that we had an error
• But we get more than that from a Hamming code
• Let’s consider an example
Hamming code example

• Data value:

0 0 1 1 1 0 1 0 0 0 1 0 1 1 0

• Add the parity bits

1 0 0 1 0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0

• That’s what you send
• Receiver checks using the same rules
• If all parity bits match, no single bit errors
What if there’s an error?

• What if a single bit is flipped?
  – Say, bit #3 changes from 0 to 1

```
1 0 0 1 0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0
1 0 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 1 1 0
```

• Now compute the parities on what you got
  – They come out to: 0 1 1 1 0
  – But in the message we have: 1 0 1 1 0
  – Error occurred!
But we get even more info

- Consider the parity bits alone
- Which bits didn’t match?

- The first and second parity bits (p1 and p2)
- Add their positions up:
  - $1 + 2 = 3$
- The error was in the third bit of the 20 bits
- So we can correct it!
Hamming

• **Error detection**
  – Detects 1 and 2 bit errors

• **Error correction**
  – Corrects all 1 bit errors
    • If more than one parity set is wrong, the parity numbers indicate the incorrect bit position
    • If only one parity set is wrong, then that parity bit itself is incorrect

• **Cost:**
  – $\log_2 N$ overhead

• **Limits:**
  – Cannot correct more than one error
Reed-Solomon

• A lot more complicated…
  – No we’re not going to walk through it

• Why interesting?
  – Add $t$ bits of overhead
  – Detects up to $t$ errors
  – Corrects up to $\left\lfloor \frac{t}{2} \right\rfloor$ errors
  – Works for burst (consecutive) errors too
Error vs. loss

- Error
  - Symbols are received, but not what was sent
- Loss
  - Nothing is received

How do you detect a loss?
The role of time

- The only way to detect loss
  - Timer expires

- Do you KNOW it was lost?
  - Nope. Maybe just late.
Dealing with loss

• So what do you do?

• At some point, assume loss occurred
  – Though perhaps it didn’t

• Then fix it!
  – In a way that won’t cause problems if you’re wrong

• ARQ: Automatic Repeat-reQuest
ARQ

• Sender
  – Transmits info in blocks with IDs
  – Keeps copies and retransmits on request
• Receiver
  – Collects blocks and looks for missing IDs
    • Typically gaps within received sequence
  – Ask sender to help *(requires reverse channel)*
    • What do you say?
    • When do you say it?
ARQ variants

Negative feedback (NACK)
- Receiver reports the IDs lost
  - Explicit request to resend
  - The IDs *presumed* lost
  - Messages could just be late
- Sender resends those explicit requests
  - No sender timers

Positive feedback (ACK)
- Receiver reports the IDs received
  - Confirms receipt
  - Implicit request to resend
  - No receiver timers
- Sender resends IDs not reported
  - Looks for gaps
  - IDs *presumed* lost
  - IDs could be lost, but so could NACK be
Variants of ARQ

• Stop-and-go
  – Positive feedback (ACK)
  – ID is 1 bit
  – Send ACK when block is received
  – Also called “alternating bit”
• Go-back-N
  – Positive feedback (ACK)
  – ID is larger
  – Send ACK when block is received
  – Sender backs up to block after (ID+1) and resends
• Selective repeat
  – Positive and/or negative (ACK/NACK)
  – ID is large
  – Sender retransmits only individual lost blocks
Reordering as error

• When is a message lost?
  – Or just late?
• What happens when messages are out of order?
  – Buffer them and reorder
  – Should this be limited? HOW?
Basic components of a channel

• A signal to use to indicate symbols
• A media the signal propagates in
• A set of symbols
• A way to generate and receive symbols
• Direction
How to create signals?

• Move photons
• Move electrons
• Move atoms
  – Motion waves (sound)
    • Pressure waves in gas, liquid
    • Transverse waves in solids
  – Streams of atoms (water flow)
• Move collections of atoms
  – Letters, flags, etc.
Types of media

• Unguided
  – Transparent
  – Mechanically conductive

• Guided
  – Transparent
  – Electrically conductive
Freespace

- Unguided
  - Transparent (includes vacuum)
  - Mechanically conductive (except a vacuum)
- Propagation velocity
  - Faster for EM
  - Slower for sound
- Signals degrade over distance
- Need a clear path
  - Not necessarily line-of-sight, though
Fibers

- Multi-mode
  - Thick core
  - Many paths
  - Many wavelengths
- Single-mode
  - Thin core
  - Fewer paths
  - Long-distance
- Hollow core
  - Uses air as the medium
  - Like freespace, but “guided”
Wires

• Guided
• Conductive
  – Material
    • Superconductors (various)
    • Silver, copper, gold, aluminum,…
• Number
  – Single-wire (ground-return)
  – Two-wire (direct return)
Communication symbols

• How we encode information on the signal
• Encodings do a lot for us
  – Represent information
  – Simplify generation
  – Simplify reception
  – Minimize errors
Some Example Encodings

• Amplitude shift keying
  – Use different signal power/strength values as symbols
• Return to zero
  – High/low signal value shows encoding
  – Signal value goes to zero between symbols
• Non-return to zero
  – Using common clock to encode/decode
• There are many others
Generating and interpreting signals

• Strictly:
  – *Generation* is a way to modulate non-varying sources to generate symbol pattern sequences that correspond to information patterns

• Practically:
  – *Generation* is a way to translate one symbol sequence into another
  – Since the source has its own representation of a sequence of symbols

• Interpretation is pretty much the same thing as generation
  – Just a different direction
Direction

• We’re still talking about 2-party communication…

• Using a single channel, which of them can send to the other?
Simplex

• A channel transfers symbols in one direction only

• How?
  – Signal propagates in a medium
**Duplex**

- A channel simultaneously transfers symbols in both directions

- **How?**
  1. Using one natively bidirectional channel and transferring non-interfering particles
     - EM, e.g., photons or RF
  2. Using two simplex channels
     - One in each direction
Half-duplex

• Introduces sharing, but still 2-party

• How?
  – Using one natively bidirectional channel
  – Ensuring that the channel always contains only symbols travelling in the same direction

• How do we ensure that?
  – Need an automated mechanism to determine which end “speaks” next
  – One element of a protocol
What Is a Protocol?

- A set of rules, agreed in advance, that enable communication
  - *Endpoints*: the things that want to communicate
  - *Link*: enables action at a distance between the two endpoints
  - *Protocol*: specifies how to automate how these interact
How Do We Automate a Protocol?

- Use a \textit{finite state machine}
  - One at each protocol participant, actually
- The “machine” is always in exactly one state
- There are a finite (and predefined) set of states
- Predefined actions cause transition from one of the states to another
Limits of FSMs

- Cannot count
  - Finite state, so limits on the count
- Cannot reverse or duplicate input
  - Duplicate would be:
    - AB -> ABAB
  - Reverse would be:
    - AB -> BA
Why do we want a FSM?

- Keep our state manageable!

- For networking, it’s enough
  – We’re basically playing “do what I do”
Mealy machine

- One type of FSM
- Has states (S)
- And transitions
  - Triggered by inputs (I)
  - Causing outputs (O)
- Generally a convenient type of FSM for networking
Sharing simple state for networking

• A wants to communicate with B
  – The goal is for A and B to share state
• Assume a perfect channel
  – No errors, loss, reordering
Simplest state

- Simplest case:
  - Two states: “round” and “funny”
  - Do the names matter?
- A decides to be in one of two states.
  - The goal of communication is for A to make B in the same state.
A gets to change state

- By itself, for some external reason

A button was pressed

It's windy

Funny

Round

(start)
B has a similar state

- Names don’t have to match

(start)

cold

sharp
B gets to change state too

- Based on what it receives
How do we communicate?

• Rules:
  – Every time A changes state, it let’s B know
  – Every time B finds out, it changes state to match
Let’s do that again, more simply

- A decides to toggle a switch UP or DOWN
  - Causes A to change state
  - Protocol makes B match A’s state
When are we done?

• When will the two states match?
  – Some time after A changes state, B will follow

• How long?
  – Who knows?
  – But it’s a reliable channel, so it WILL change state eventually
  – Can we do better than that?
Mutual state

- States of A and B:
Mutual state

- States of A, B, A’s view of B, B’s view of A:
Mutual state

• Keep going!
  – No limit to the mutual modeling
Limiting mutual state

• Stop at one step
  – Your state
  – Your view of the other end’s state
Simple communication with confirmation

- Still sending info just from A to B
  - A models both sides
  - B confirms when it has changed state
Confirmation

• How does A know B learned of the state change?
• Positive acknowledgement
  – ACK
  – Confirms receipt of information
Complication #1: imperfection

- Real channels aren’t perfect
- Loss
  - Need to handle that
- Error
  - That can happen, too
  - But detect and address it as loss
  - Error you don’t detect isn’t an error (!)
    - It’s the definition of your system . . .
How do we detect loss?

• Is it lost or just late?
• We can never know for sure
• We can only give up
  – When timer expires, we declare “loss”
Simple communication with loss

• Still sending info just from A to B
  – Add repeats based on timeouts

• But what if an ack is lost?
Time out on ACKs, too

• The three-way handshake (TWHS)
What’s magic about TWHS?
What’s magic about TWHS?

• Both sides have confirmed with each other

• We’ve achieved our limited mutual state
Specifying a protocol

- States
  - Endpoint values
- Symbols
  - Messages “on the wire”
- Events
  - Incoming
  - Outgoing
- Transition table
  - Relates the above
- All expressible as a state machine
Let’s break that down a bit more

• States at the endpoints
• Symbols “on the wire”
• Events
  – IN:
    • Symbols received from the channel (receive)  
      Unix receive()
    • Symbols incoming at the sender  
      Unix write()*
    • Timer expires
  – OUT:
    • Symbols sent on the channel (transmit)  
      Unix send()
    • Symbols out from the receiver  
      Unix read()*
    • Timer is set
• Transition table
  – Maps events and states to other events and new states

* these are used from outside the protocol, so write() is when an external process sends data into the protocol
TCP state diagram

- Remember
  - They’re just NAMES
  - It’s the relation, not the name, that has meaning
This should look familiar

- **What have we seen before?**
  - Two sets of handshakes
- **What’s the rest?**
  - “corner cases”
Complication #2: sharing complex state

• What if we want to share more than one bit?
• Share bulk by “leap-of-faith”
  – Share a sequence of states
What’s the leap of faith?

• We already know how to share a bit
• To share more:
  – First we agree on a first bit
    • We’re both on the call
  – Then we agree on each block of bits sent
    • We’re really agreeing on one bit:
      – Did you get that block?
      – “That” defined by an ID (sequence number) and checksum
  – Finally we agree we’re done
  – We assume that if the above sequence is true, then the file was communicated correctly
Sequence of states

- Step through N items
  - Move forward once confirmed
  - Stepwise agreement
  - End with a final agreement
This 2-party stuff seems universal

• It is!
  – All protocols should be described the same way
    • States
    • Symbols (message formats)
    • Events
    • Transition tables
  – State diagrams have familiar parts
    • Three-way handshake
    • Confirmed shared state
Look at TCP

• **States**
  - Connection status
    • CLOSED, LISTEN, SYNSENT, SYNRECD, ESTABLISHED, …
    • Round-trip time, congestion window, …
  - Blocks transferred
    • Sequence number

• **Symbols**
  - SYN, FIN, RST, ACK
  - Block sequence ID
  - Data with checksum

• **Events**
  - **Input**
    • Message arrivals
    • Timer expires
    • Write events
  - **Output**
    • Message departure
    • Timer to set
    • Read events

• **Transition table**
  - State + inputevent -> newstate, outputevent
Why is it hard?

• Protocols can be large
  – Made of familiar parts
  – But many such parts

• There’s more than 2 parties to consider
  – And we’re getting to that soon too…
Summary

• Even noisy channels are useful
  – And we can calculate exactly how useful

• Errors happen
  – We can detect them
  – We can correct them

• We use protocols to automate communication
  – Which are implemented with finite state machines