

Retrospective: Computer System Design

Hung-Wei Tseng

What is end-to-end argument? Can you identify a few examples of designs using end-to-end arguments

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

If you can't do it properly, don't do it at all.

Practices of End-to-end arguments

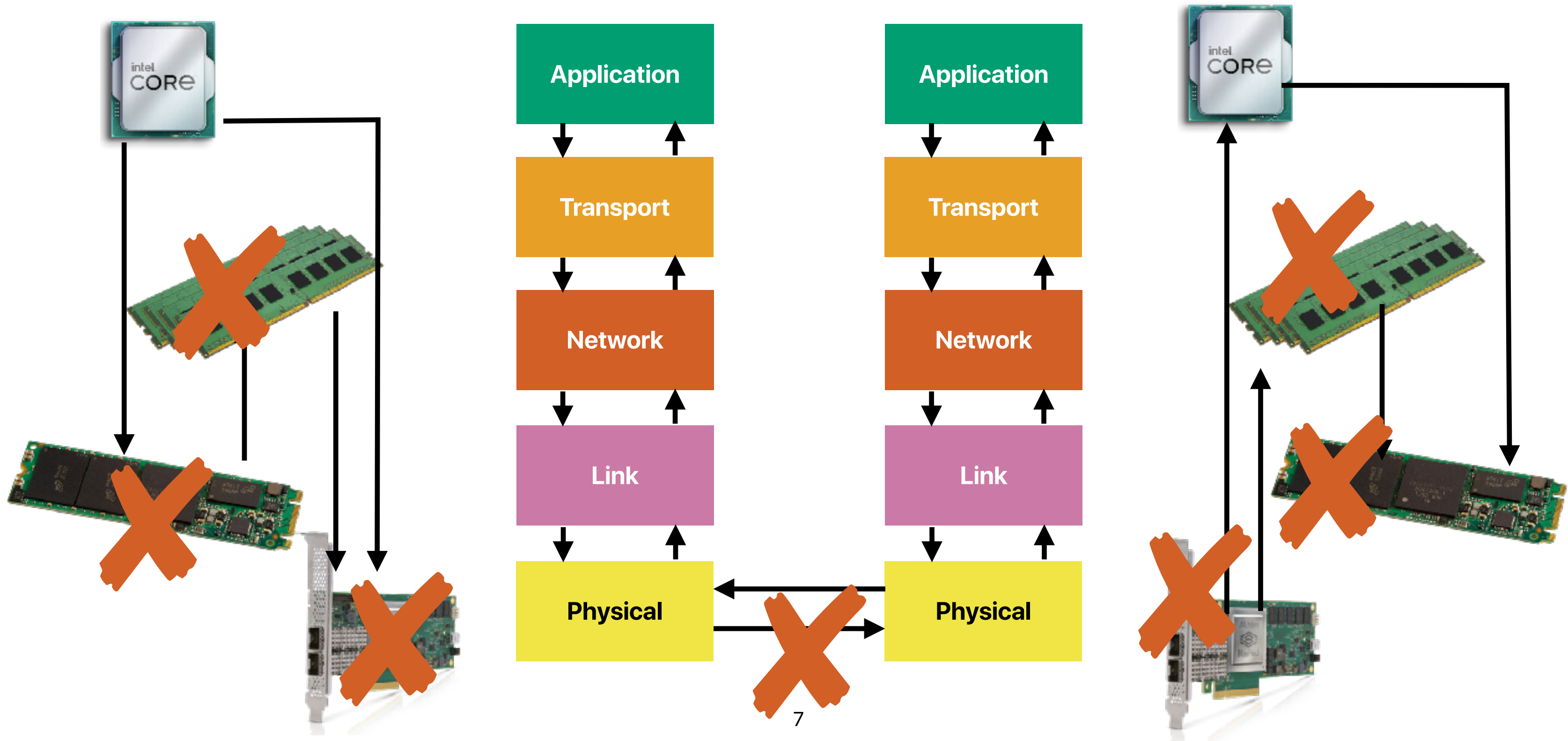
Practices of End-to-end arguments

- Encryption
- Transactions
- Cellular: human retry

End-to-end arguments in system design

**J. H. Saltzer, D. P. Reed, and D. D. Clark.
MIT**

File transfer between two computers



How to guarantee the correctness?

- Point-to-point
 - Check the correctness of data in each step
 - Make sure the operation is completed by sending "ACK"s to the last step
 - Retry from the last step when there is an error or timeout
- End-to-end
 - Create a checksum at the application level/file system
 - Compare the checksum at the receiver
 - Resend the request if failed
- Or both
 - Point-to-Point checks
 - End-to-End checks must still be performed, since only one of the threats is handled

Why End-to-End in communication systems

- Why?
 - Routers knows nothing about end host processes
 - The network is simply incapable of providing process-to-process communication.
- Cost of extra functionalities can be penalties
 - Not all applications want in-order guaranteed delivery service, e.g. video streaming.
 - And really, applications don't care if a packet is guaranteed to be delivered from one router to another. They want end-to-end guaranteed delivery.

**In modern Internet, what approach
do we take? Why?**

Packets in modern communication networks




TCP segment header																																	
Offsets		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0 0				C W R	E C N	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if data offset > 5. Padded at the end with "0" bits if necessary.)																															
:	:																																
56	448																																

802.3 Ethernet packet and frame structure									
Layer	Preamble	Start frame delimiter (SFD)	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap (IPG)
Length (octets)	7	1	6	6	(4)	2	42 - 1500 ^[6]	4	12
Layer 2 Ethernet frame	(not part of the frame)		← 64 - 1522 octets →						(not part of the frame)
Layer 1 Ethernet packet & IPG	← 72 - 1530 octets →							← 12 octets →	

Why not e2e?

- The link is unreliable, the performance overhead of retry is significant if the packet goes through many network links




Which side follows the end-to-end argument?

 Please translate the following code into RISC-V instructions 
`uint_xlen_t pcnt(uint_xlen_t rs1) {
 int count = 0;
 for (int index = 0; index < XLEN; index++) {...`


✦ Here's the equivalent RISC-V Assembly code for the `pcnt` function, assuming the Bitmanip extension is not available:

Code snippet

```
pcnt:  
    # Set register x0 to zero (count variable)  
    li x0, 0  
  
    # Loop XLEN times (assuming XLEN is a macro for register size)  
loop:  
    # Shift rs1 right by the index (assuming index is in register x1)  
    srl x2, x1, x1 # x2 = rs1 >> index  
  
    # Bitwise AND x2 with 1 to get the least significant bit  
    andi x2, x2, 1 # x2 = x2 & 1  
  
    # Add x2 to x0 (count)  
    add x0, x0, x2 # count = count + x2  
  
    # Add 1 to index register  
    addi x1, x1, 1 # index = index + 1  
  
    # Check if index is less than XLEN, loop if true  
    blt x1, xlen, loop # if (index < XLEN) goto loop  
  
    # Return the count in x0  
    ret
```

 Please translate the following code into x86 using `popcnt` instruction 
`uint_xlen_t pcnt(uint_xlen_t rs1) {
 int count = 0;
 for (int index = 0; index < XLEN; index++) {...`


✦ Here's the equivalent x86 assembly code for the `pcnt` function using the `popcnt` instruction:

Code snippet

```
pcnt:  
    # Move rs1 (argument) to eax  
    mov eax, DWORD PTR [rsp + 4] # Assuming rs1 is the first argument  
  
    # Use popcount to directly count set bits in eax and store in ecx  
    popcnt ecx, eax # ecx = popcount(eax)  
  
    # Return the count in ecx  
    ret
```

Use code [with caution.](#) 

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system.

Is the trend of accelerator-rich/near-data architectures an end-to-end design? Why do we or why don't we follow the end-to-end design?

Hardware accelerators & NDP

- Like CISC (e.g., x86), we make the device more intelligent
- Performance
- Drawbacks
 - Are we able to reflect the latest demand of computation?
 - Are we able to expose the features for flexibility?

Hints for computer system design

Butler W. Lampson

Computer Science Laboratory Xerox Palo Alto Research Center

Hints for computer system design

Why?	Functionality Does it work?	Speed Is it fast enough?	Fault-tolerance Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load End to end Safety first	End to end
Interface	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Do you think the adoption of hardware-accelerators/ NDP is good for computer system designs?

Why?	<i>Functionality</i> Does it work?	<i>Speed</i> Is it fast enough?	<i>Fault-tolerance</i> Does it keep working?
Where?			
<i>Completeness</i>	Separate normal and worst case	Shed load End to end Safety first	End to end
<i>Interface</i>	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
<i>Implementation</i>	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Accelerators/NDP vs. Hints for computer system design

Why?	<i>Functionality</i> Does it work?	<i>Speed</i> Is it fast enough?	<i>Fault-tolerance</i> Does it keep working?
Where?			
<i>Completeness</i>	Separate normal and worst case 😊	Shed load End to end Safety first	End to end
<i>Interface</i>	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
<i>Implementation</i>	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Completeness

- Separate **normal** and worst case
 - Normal means the most time consuming — it is normal for 80% of the time to be spent in 20% of the code
 - Amdahl's Law
 - Hardware accelerators improve the **normal** case
- Make normal case fast
 - Amdahl's Law
- The worst case must make progress
 - That's why we still need general-purpose processors

Accelerators/NDP vs. Hints for computer system design

Why?	Functionality Does it work?	Speed Is it fast enough?	Fault-tolerance Does it keep working?
Where?			
Completeness	Separate normal and worst case 😊	Shed load End to end Safety first	End to end
Interface	Do one thing well: Don't generalize 😊 Get it right Don't hide power 😊💧 Use procedure arguments 😊💧 Leave it to the client 😊💧 Keep basic interfaces stable 😊💧 Keep a place to stand 😊💧	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Interface — Keep it simple, stupid

- Do one thing at a time or do it well
 - Don't generalize — Make it fast, rather than general or powerful.
 - It is much better to have basic operations executed quickly than more powerful ones that are slower (of course, a fast, powerful operation is best, if you know how to get it).
 - RISC v.s. CISC?
- Get it right
 - What about approximate computing?

More on Interfaces

- Don't hide power — higher levels should not bury this power inside something more general
 - Are we doing all right with FTL?
 - Interface for hardware accelerators?
 - How does the accelerator quantize data internally?
- Use procedure arguments to provide flexibility in an interface
 - Thinking about SQL v.s. function calls
- Leave it to the client — as long as it is cheap to pass control back and forth
 - General-purpose programmability still matters
 - Is passing the control back and forth really cheap?

Continuity

- Keep basic interfaces stable
 - What happen if you changed something in the header file?
 - Tensorflow v1.0 vs. Tensorflow v2.0
- Keep a place to stand if you do have to change interfaces
 - Mach/Sprite are both compatible with existing UNIX even though they completely rewrote the kernel

Accelerators/NDP vs. Hints for computer system design

Why?	Functionality Does it work?	Speed Is it fast enough?	Fault-tolerance Does it keep working?
Where?			
Completeness	Separate normal and worst case 😊	Shed load End to end Safety first	End to end
Interface	Do one thing well: Don't generalize 😊 Get it right Don't hide power 😊 Use procedure arguments 😊 Leave it to the client 😊 Keep basic interfaces stable 😊 Keep a place to stand 😊	Make it fast Split resources Static analysis Dynamic translation 😊 😊 😊	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets 😊 Use a good idea again 😊 Divide and conquer 😊	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Making implementations work

- Plan to throw one away
- Keep secrets of the implementation — make no assumption other system components
 - Don't assume you will definitely have less than 16K objects!
 - What rules hardware accelerators break?
- Use a good idea again
 - Caching!
 - Replicas
 - Logs
- Divide and conquer
 - Matrix tiling
 - Parallel processing
 - Map-reduce

Accelerators/NDP vs. Hints for computer system design

Why?	Functionality Does it work?	Speed Is it fast enough?	Fault-tolerance Does it keep working?
Where?			
Completeness	Separate normal and worst case 😊	Shed load 😊 End to end Safety first	End to end
Interface	Do one thing well: Don't generalize 😊 Get it right Don't hide power 😊💧 Use procedure arguments 😊💧 Leave it to the client 😊💧 Keep basic interfaces stable 😊💧 Keep a place to stand 😊💧	Make it fast 😊 Split resources 😊 Static analysis 😊 Dynamic translation 😊	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets 😊 Use a good idea again 😊 Divide and conquer 😊	Cache answers 😊💧 Use hints 😊 Use brute force 😊 Compute in background Batch processing 😊	Make actions atomic Use hints

Speed

- Split resources in a fixed way if in doubt, rather than sharing them
 - Hardware accelerators don't support multiprogramming very well
- Use static analysis — compilers
 - Thinking about XLA, CUDA
- Dynamic translation from a convenient (compact, easily modified or easily displayed) representation to one that can be quickly interpreted is an important variation on the old idea of compiling
 - PyTorch
 - Tensorflow

Speed (cont.)

- Cache answers to expensive computations, rather than doing them over
 - Consistency of caching would hurt the performance again
 - Hardware accelerator/heterogeneous computing makes caching difficult
- Use hints to speed up normal execution
 - Prefetching
 - Branch prediction

Speed (cont.)

- When in doubt, use brute force
 - Especially as the cost of hardware declines, a straightforward, easily analyzed solution that requires a lot of special-purpose computing cycles is better than a complex, poorly characterized one that may work well if certain assumptions are satisfied
 - Ken Thompson's chess machine Belle relies mainly on special-purpose hardware to generate moves and evaluate positions, rather than on sophisticated chess strategies.
 - Even an asymptotically faster algorithm is not necessarily better. There is an algorithm that multiplies two $n \times n$ matrices faster than $O(n^{2.5})$, but the constant factor is prohibitive.
 - The 7040 Watfor compiler uses linear search to look up symbols; student programs have so few symbols that the setup time for a better algorithm can't be recovered.

Speed (cont.)

- Compute in background when possible
 - Free list instead of swapping out on demand
 - Cleanup in log structured file systems: segment cleaning could be scheduled at nighttime.
- Use batch processing if possible
 - Soft timers: uses trigger states to batch process handling events to avoid trashing the cache more often than necessary
 - Write buffers
 - Batch processing in modern ML applications
- Safety first
- Shed load to control demand, rather than allowing the system to become overloaded

Accelerators/NDP vs. Hints for computer system design

Why?	Functionality Does it work?	Speed Is it fast enough?	Fault-tolerance Does it keep working?
Where?			
Completeness	Separate normal and worst case 😊	Shed load 😊 End to end Safety first	End to end
Interface	Do one thing well: Don't generalize 😊 Get it right Don't hide power 😊 Use procedure arguments 😊 Leave it to the client 😊 Keep basic interfaces stable 😊 Keep a place to stand 😊	Make it fast 😊 Split resources 😊 Static analysis 😊 Dynamic translation 😊	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets 😊 Use a good idea again 😊 Divide and conquer 😊	Cache answers 😊 Use hints 😊 Use brute force 😊 Compute in background Batch processing 😊	Make actions atomic Use hints

Fault Tolerance

- An under-developed area in hardware accelerators/near-data processing
- Completely end-to-end — restart the computation if the end user don't feel it right
- Log updates
 - Checkpointing for now
 - Can we make use of persistent memory?
- Make actions atomic or restartable
 - What happens if the accelerator becomes faulty?

Conclusion

- Any system, not just data-centric computing system, should care the end-to-end performance/correctness
- We have learned tools to identify the performance bottlenecks — the roofline model
- We have learned new rigs in addressing the performance bottlenecks
 - Hardware accelerators
 - Processor-in/using-memory
 - Near-data processing
- System design is challenging
 - Programmability
 - Performance
 - Security

Announcement

- Final project presentation — 3/21 11:30a-2:30p (SSC 125)
 - Please book your order on the calendar <https://calendar.google.com/calendar/u/0/selfsched?sstoken=UURHeVJHWDhIUxhhfGRIZmF1bHR8ZGQ5ODhhOGRhYjU1NGRjOWY1ODM3MmZiZjlmNmE0ODU>
 - Pizza provided

Electrical Computer Science Engineering

277

Finale

