# Coding for High Frequency Trading and other Financial Services applications

Richard Croucher

March 2017

BARCLAYS

BARCLAYS

# About me

Richard is currently Vice President of High Frequency Engineering for Barclays

As well as Barclays, Richard has consulted on IT to HSBC, RBS, DeutscheBank, CreditSuisse, Flowtraders, JP. Morgan, Merril Lynch and Bank of America.

Richard was also Chief Architect at Sun Microsystems where he worked on HPC Grid and helped create their Cloud capability. He was also a Principle Architect in Microsoft Live which evolved in Azure.

Functionally he has help positions as a Physicist, Electronic Design Engineer, Programmer, IT Product Manager, IT Marketing Manager, IT Consultant and IT Architect

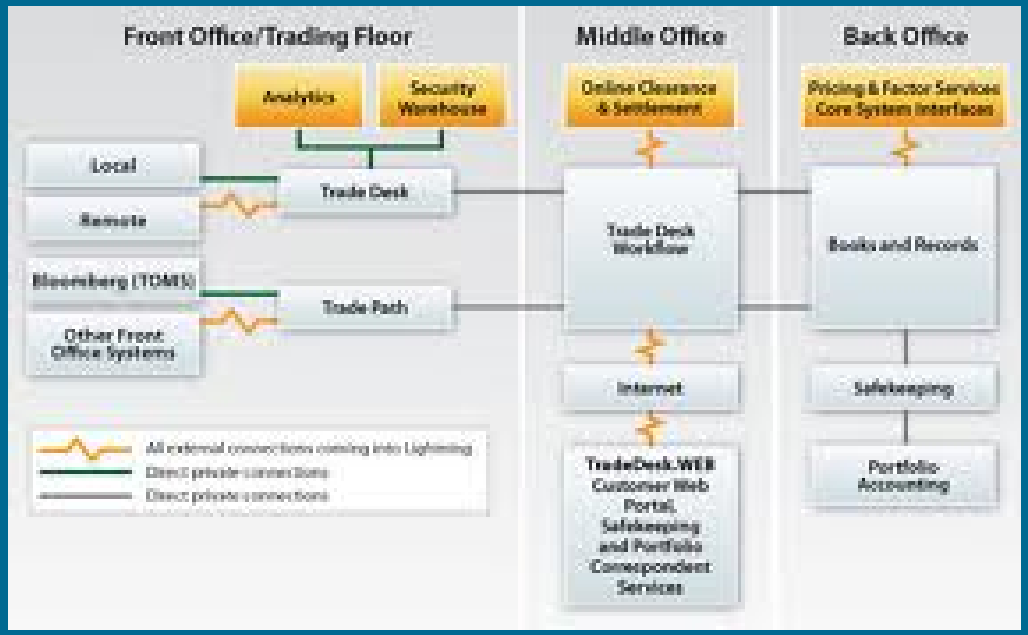
Describes himself as a Platform Architect, specialising in HFT, DevOps, Linux and large scale Cloud solutions

Fellow of STAC Research, Fellow British Computer Society, Chartered IT Practicioner. Awarded degrees from Brunel University, University of East London and University of Berkshire

# Intro

- Investment Banking IT
- High Frequency Trading
- Common Technologies
- Coding Styles

# Investment Banking IT




Front Office

- Traders sit on Trading rooms, each with several screens and a fast dialler.
- Complimented now by cololocated algo trading computers

Middle Office

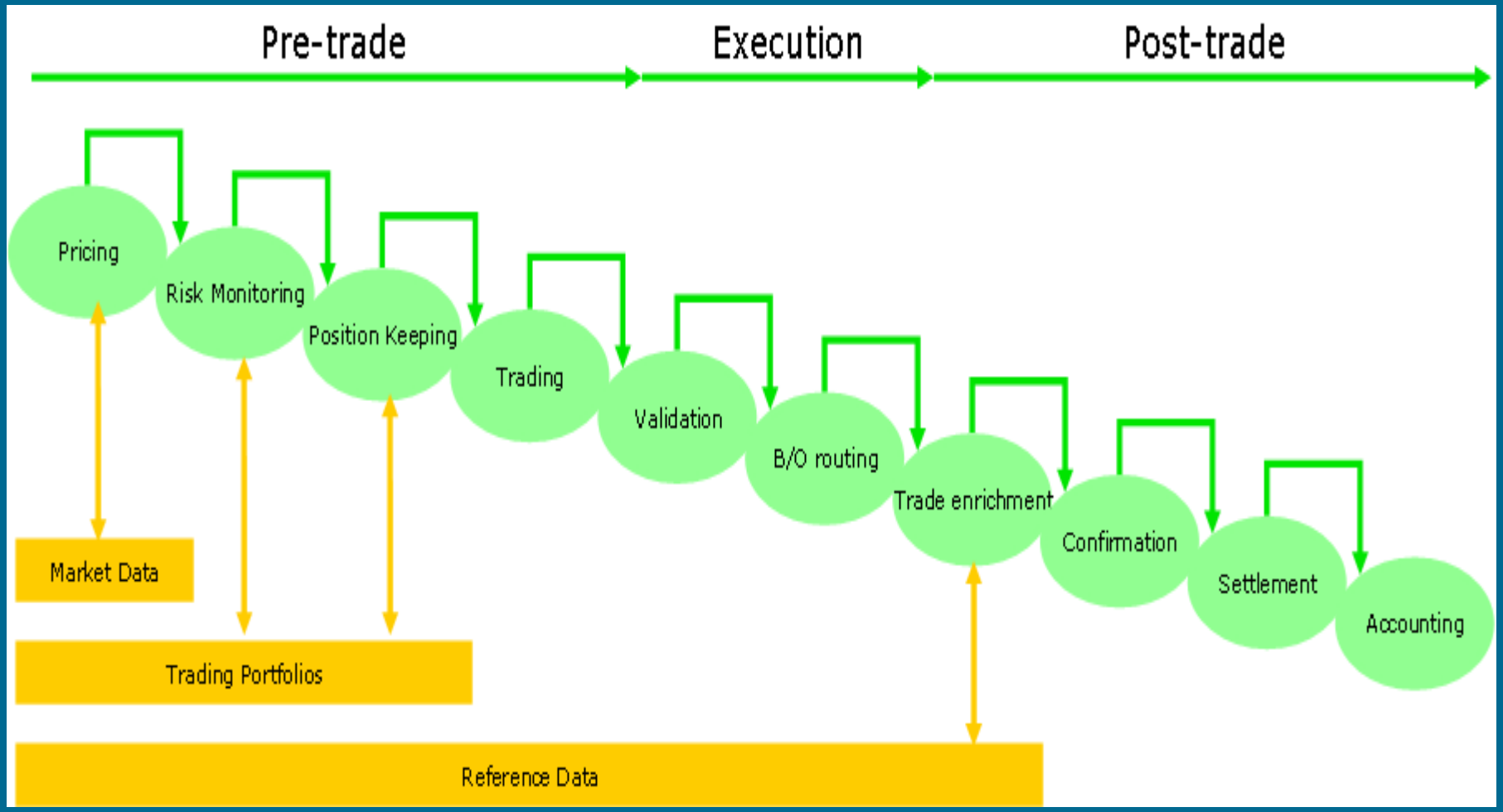
- Accountants, Economists, Mathematicians create strategies, watch exposure and risk on client portfolios.
- Trades are matched, cleared and settled

Back Office

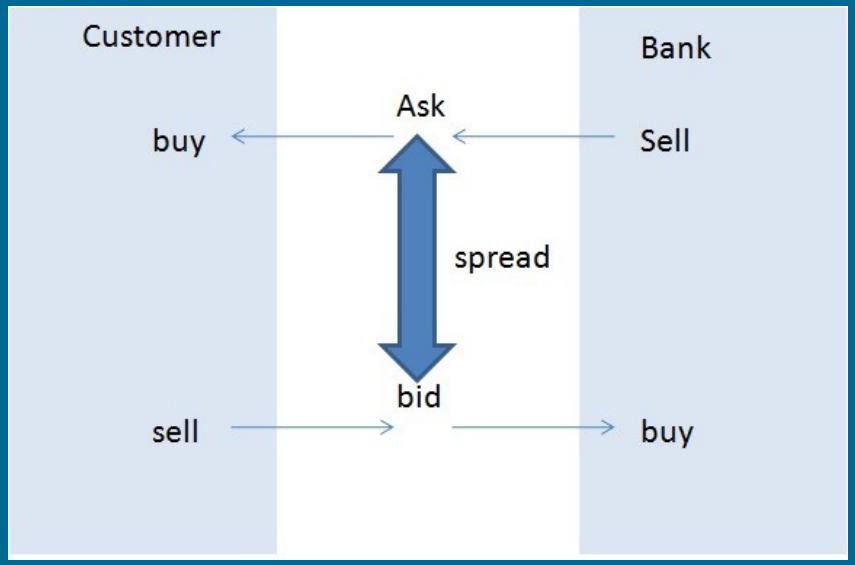
- The day's cash movements are aggregated and payment instructions sent out
- The Banks overnight positions are re-calculated and updated

BARCLAYS

# Trade Flow


Pre-trade
Execution
Post-trade
Pricing
Risk Monitoring
Position Keeping
Trading
Validation
B/O routing
Trade enrichment
Confirmation
Settlement
Accounting
Market Data
Trading Portfolios
Reference Data

# Trading - It’s all buying and selling

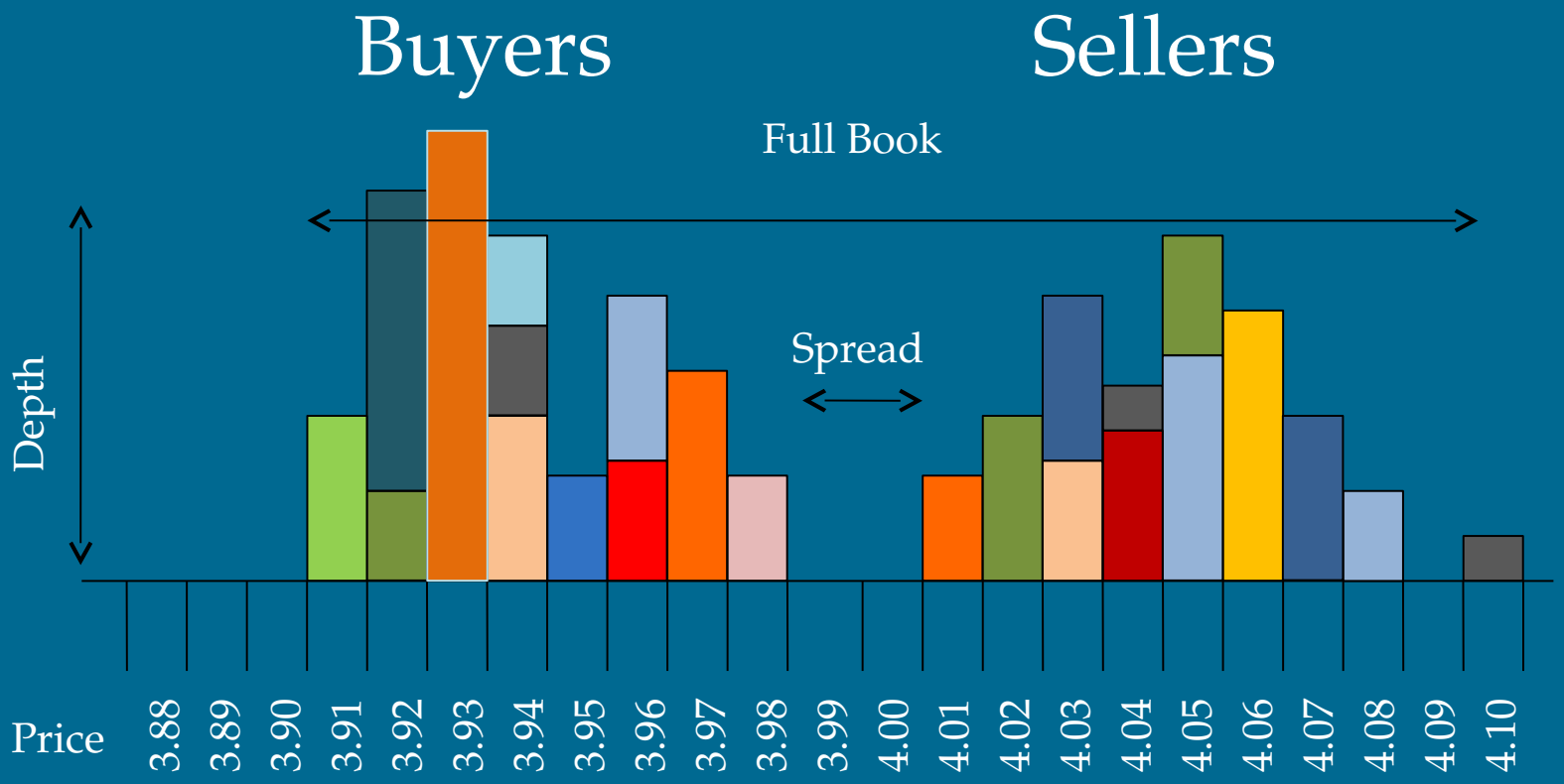



The **venue** is the electronic meeting place where buyers and sellers get connected
The **venue** matches buyers to sellers
The **spread** is the current difference between buy and sell offers

Liquidity increases as buyers and sellers use your venue

Options are contracts to buy/sell in the future at an agreed price

# Trading book




- Orders can be filled when they match buyers to sellers at the same price
- Orders typically stay until cancelled or the market closes

# Algorithmic Trading Strategies

**Arbitration**
- Exploiting difference in price for the same stock in different venues

**Momentum**
- Assumes that a current trend will continue

**Alpha (pairs)**
- Matches stocks and assumes the value of one should be the same as another
- Based on fundamental macroeconomic statistics

**Composites**
- Arb a ETF by beating it's update,  e.g. A 10% change in BP value on the FTSE100 may translate into a 0.2% change in the overall index

**Market Making**
- Accepting an Exchange fee to provide liquidity
- Typically large spread just outside current price
- Object is to make pennies  on volume and minimize holding

# Buying strategies

**Smart Order Routing**

- Discover which venue is offering the best price
- A regulatory requirement in EU and USA

**Iceberg**

- Slice a large order up into lots of small orders to disguise intent and reduce market impact
- Led to a big increase in order volumes and decrease in typical order size

**Sweep Order**

- Buy only a percentage of desired quantity and pause for more liquidity to be placed, hopefully at the same price

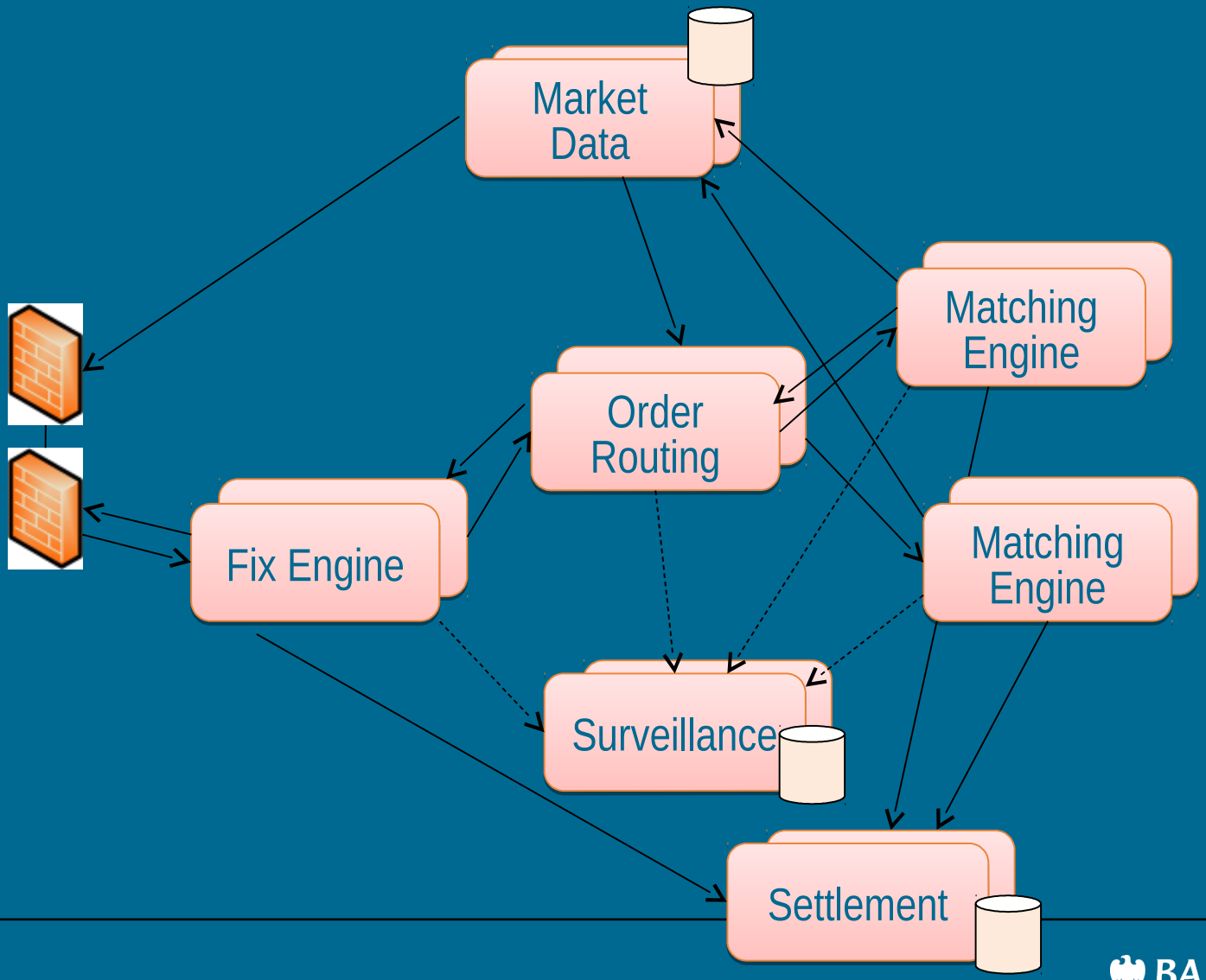
**Crossfire**

- Liquidity capturing algorithm that targets liquidity within dark pools

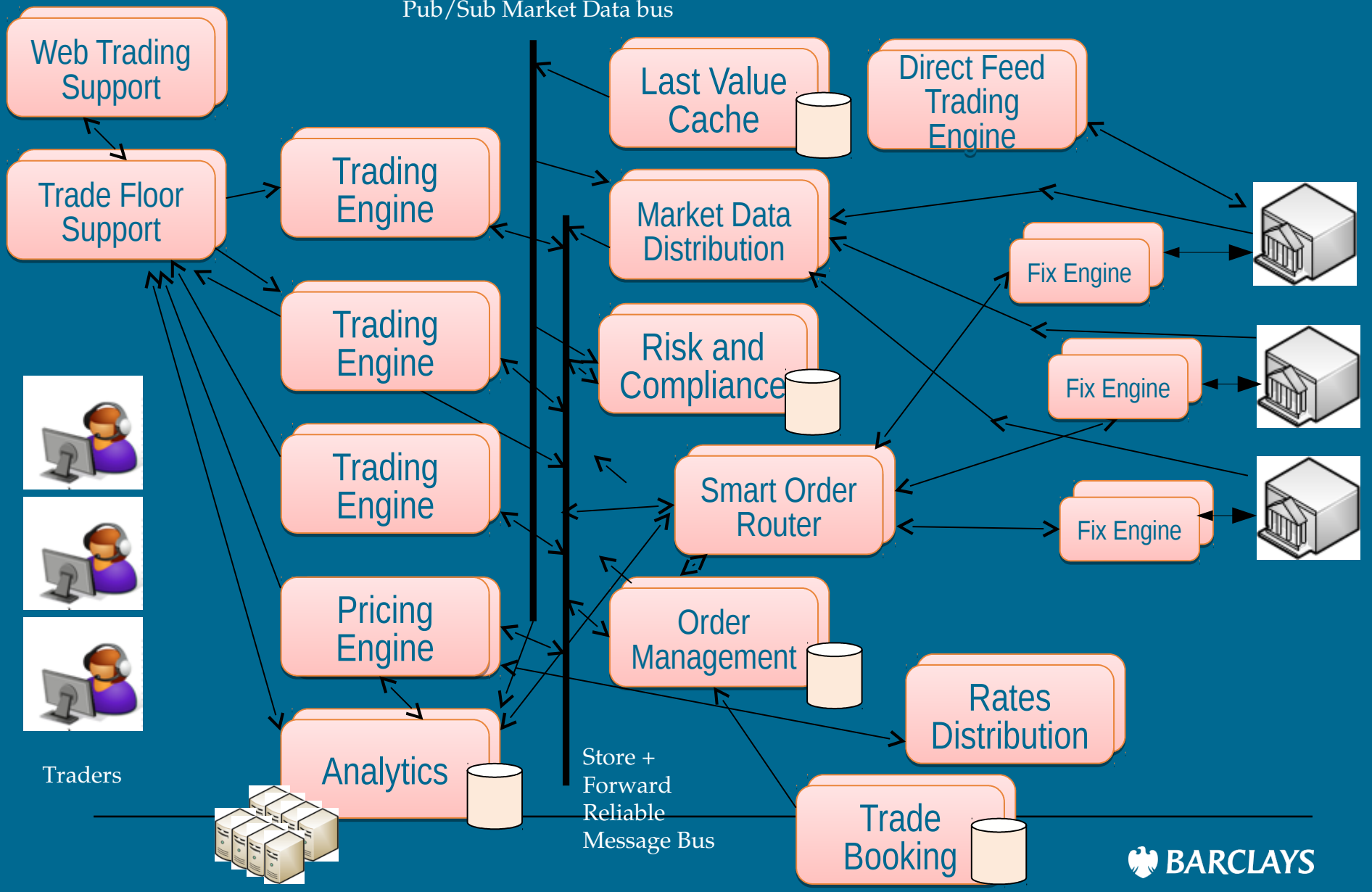
# Common Technologies Deployed

- Time Series Database – tick data - mostly kdb
- In memory Database
- Real time analytics  - Hadoop, Shark
- Excel analytics and Grid compute plugins
- Complex maths libraries
- Packet Decoders for each venue and trading protocol
- FIX Engines
- Smart Order Routers
- RDMA  (Remote Direct Memory Access)
- FPGA   (Fuse Programmable Gate Array)
- DevOps -  Chef, Puppet

# Sell Side System - Venue



# Buy Side System



# QUANT programming

Now generally used to refer  to the development of algorithmic trading systems
Heavy maths bias – Ph.D expected
Expect familiarity and understanding of Black-Scholes model used for derivatives
    e.g.  The value of a call option for a non-dividend paying underlying stock is

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right]$$

$$d_2 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)(T-t)\right]$$

$$= d_1 - \sigma\sqrt{T-t}$$

The price of a corresponding put option based on put-call parity is:

$$P(S,t) = Ke^{-r(T-t)} - S + C(S,t)$$

$$= N(-d_2)Ke^{-r(T-t)} - N(-d_1)S$$

# HFT – programming skills

C++ and Java dominate,  with a small number of FPGA specialists

- Packet processing – TCP, UDP, Multicast
- Destructor threads – spin waiting for packets to arrive to avoid interrupt wakeup delay
- Actor model – minimise lock contention
- Nanosecond timestamping
- Direct  buffer  management
- Warm up -  allocate and preload everything before trading starts
- Pinning memory,  cache line alignment
- CPU isolation and affinity
- Achieving durability via replication across network rather than disk write (often to ‘luke warm’ backup server)
- Memory mapping files when writing to disk cannot be avoided

# C++ expertise area’s

- Boost – particularly Math
- C and even assembler inline
- QuantLib - Greeks library
- Lockfree++, actor patterns
- Performance optimization with pragma’s
- Intel TBB (Thread Building Blocks)
- Code and data locality to reduce cache misses
- Compiler optimization
- Network processing – TCP, UDP, Multicast
- TCP bypass - RDMA - libibverb
- Memory optimization and tuning – cache alignment, huge pages
- Tuning – Intel Vtune

# Java expertise area

- Low latency tuning and jitter avoidance
- Extensive use of  NIO, particularly with buffers
- Lock detection, avoidance and tuning
- Network processing -  TCP,  UDP, Multicast
- Packet processing -   raw,  pcaps
- RDMA – IBM JSOR, NIO wrappings for libibverb
- GC tuning and avoidance  - explicit buffer management and re-use,  tuning -
  - Numa aware on multi-socket servers  -XX:+UseNuma
  - Reduce the cache misses  -XX:+UseCompressedOops
  - Reduce the amount of TLB misses  -XX:+UseLargePages
  - Increase object persistence -  -XX:MaxTenuringThreshold=4

# Linux

Virtually all trading carried out on Linux

- Goal is to achieve Low latency and low jitter
- Programmers are expected to know about the techniques used since there are implications in the code
- Constant battle with power saving features added to each new processor and Linux release - C and P states
- Isolating cores from scheduler and then explicit core selection for critical threads - sched_setaffinity(2)
- Turbo boost,  overclocking
- Kernel bypass preloads  - Solarflare Onload,  libvma, speedus
- TCP bypass - RDMA -  RoCE, InfiniBand, OmniPath
- Linux bypass - Data Plane Development Kit (DPDK)
- Performance monitoring and profiling tools – sar, perf, iostats, mpstat, memprof, strace, ltrace, blktrace, valgrind, latencytop, systemtap ....

# FPGA Programming

Mix of hardware and software skills

1. Hardware selection – device, board
2. Design and Code - VHDL Verilog
3. Simulation
4. Synthesis
5. Test bed instantiation
6. Pin assignment
7. FPGA bitstream generation and program load
8. Test

See Sven Anderssons “How to design an FPGA from scratch” published in EE Times, good place to start although dated

# Multicore

Servers supporting 1000 hardware threads are already available. Trend will increase with Moore's law doubling this every 18 months

Scalability, particularly concurrency is too hard with imperative programming languages

Functional Languages are a better match to create scalable code  to run on multiple cores

Functional languages implicitly better for event based, lambda programming styles

# Functional Languages - Erlang

Benefits of Erlang

- Erlang OTP provides comprehensive runtime support including - Debugger, Event Managers, Watchdogs, FSM, in-memory DB, Distributed DB, HA, Unit test, Docs, Live Update
- Big **Int** – no need to deal with overflows
- Built in support for HTML and SNMP
- Powerful bit level processing
- Code is more powerful – achieves more in fewer lines
- Automated restart using Supervisors – fail early strategy significantly reduces explicit try/catch coding
- Vast ecosystem of Erlang code, particularly for messaging and comms

Challenges with Erlang

- Immutable variables take some getting used to
- Forces you to use recursion since no 'while', 'for' operations
- Native String handling inefficient - text intensive systems use bit strings
- Overhead of typed data reduces efficiency e.g. Int's
- Virtual Machine, GC, Interpretation overheads although HiPE provides optimized support for Unix on x86.

Practical experience is that inherent concurrency more than compensates for VM overhead for most real work.
Exceptions are intensive numeric calculation and ultra low latency trading

# Erlang distributed computing – ping/pong

```
ping(0, Pong_PID) ->
        Pong_PID ! finished,
        io:format(“ping finished~n”, []);
ping(N, Pong_PID) ->
        Pong_PID ! {ping, self()}
        receive
                pong ->
                        io:format(“Ping received pong~n”, [])
        end,
        ping(N -1, Pong_PID).

pong() ->
        receive
                finished ->
                        io:format(“Pong finished~n”, []);
                {ping,  Ping_PID} ->
                        io:format (“Pong received ping~n”, []),
                        Ping_PID ! pong,
                        pong()
        end.

start() ->
        Pong_PID = spawn(ping_pong, pong, []),
        spawn(Server2, ping_pong, ping, [3, Pong_PID]).
```

# Erlang Foreign Language integration

OS cmds:

Function os:cmd executes the command and returns the result. Exposes dependencies on runtime operating system.

Ports:

Emulates Erlang node, separate failure and scheduling domain, safest but highest overhead. Built in support for ‘C’ (erl_interface lib) Java (jinterface). Community available OTP.NET, Py-interface (Python), Perl, erlectricity (Ruby), PHP, Haskell/Erlang-FFI, Erlang/Gambit (Scheme), Distel (Emacs Lisp), Rustler (Rust)

Linked-in drivers:

Runs inside a Erlang node. Dynamically linked in at runtime. Logically behaves like a port but with less overhead. Shares failure domain. A crash of one will crash both. Need to use driver_alloc(), driver_free() for malloc.

NIFs:

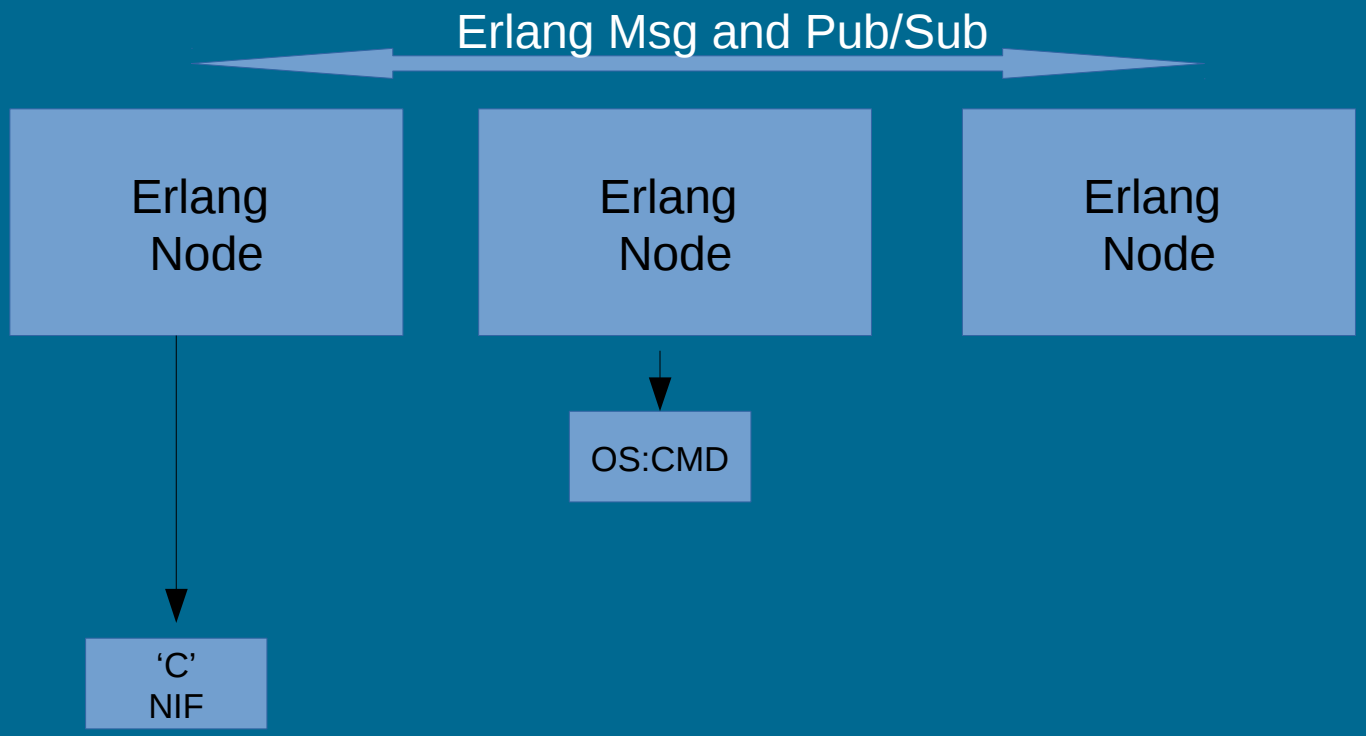
Replaces Erlang function, shares failure domain and thread scheduling. Can impact node scheduling if execution > 1-2mS, needs to explicitly pre-empt by yielding. Setup enif_schedule_nif so that node restart the still to complete NIF

# Erlang HFT example

Orchestration
Control Plane
(mult-threaded)

Linux
setup/config

High
performance
task
single-
threaded)



# Advanced Networking

Trading Venues are at extremely high volumes and speeds  e.g OPRA 30 million msg/sec  -  40G Ethernet

Extensive use of Multicast to  ensure all participants receive data at same time

- Linux treats UDP as disposable so constant battle with ‘drops’
- Needs explicit support in network  - disabled in AWS.
- TCP/IP Single thread performance limited to around 15Gbps
- User space TCP/IP increase this to about 20Gbps but still too slow
- RDMA is required to achieve single threaded line speed for interface > 10G
- Read and write to remote memory at line speed and without consuming CPU cycles on remote node
- Most 25/40/100G NICs all support RDMA
- Variants of latest Intel and AMD server CPU’s have RDMA on die
- InfiniBand,  Ethernet (RoCE) and OmniPath transports all unified by libibverbs

# Programming with RDMA VERBS

Four phases to a RDMA program

- Connection management and establishment
  - rdma_cm allows TCP/IP address space to be used to establish ‘queue pairs’ which are used as end points.
- Memory registration
  - locks the memory region which will send or receive data into memory to prevent page faults. HCA loads TLB’s to translate between virtual to physical address
  - exchange memory keys to permit remote access to this memory by the key holder
- Data transfer
  - queue a work request , e.g. Read from or write to a remote server
    - Single block or scatter gather
  - actual data transfer carried out by hardware at wire speed
    - Up to 2GB in a single transfer, HW error detection and correction prevents errors
    - measured at 6Gbytes per second, < 2µS latency across network for 2KB transfers
- Completion
  - Receive or check for notification of completion

Questions?

Richard.croucher@Barclays.com