

Bijlage VII

“Pipeline”

Student:	Jasper Lammers
Student nummer:	2108351
Begin studie:	2007
Afstudeer periode:	07-02-2011 t/m 24-06-2011
Opleiding:	Technische Informatica, voltijd
Bedrijf:	Memoria Media
Locatie:	Eindhoven
Bedrijfsbegeleider:	Erwin Wullems, Algemeen Directeur
Technisch begeleider:	Henk Jooren
Docentbegeleider:	Peter Lambooi
Date:	07-06-2011
Document release:	01
Document status:	Finished

Inhoudsopgave

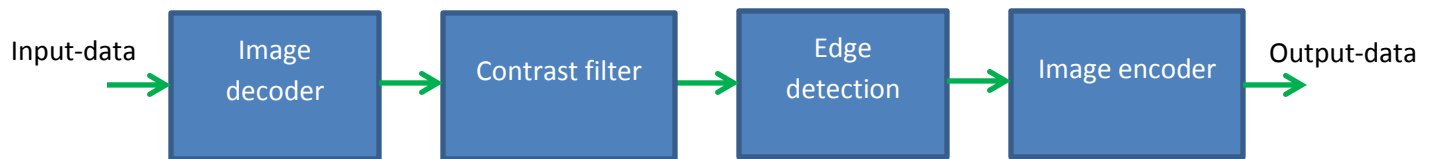
1. Inleiding.....	2
2. Composite patroon	3
3. Multithreading	4
4. Queue.....	5
4.1 Priority.....	6
4.2 Throttling	6
5. Processing time	7
6. Distributed computing.....	8
6.1 Inleiding.....	8
6.2 Pipeline en distributed computing.....	8
7. Meerdere inputs/outputs	9
8. Conclusie.....	10
8.1 Voordelen.....	10
8.2 Nadelen	10

1. Inleiding

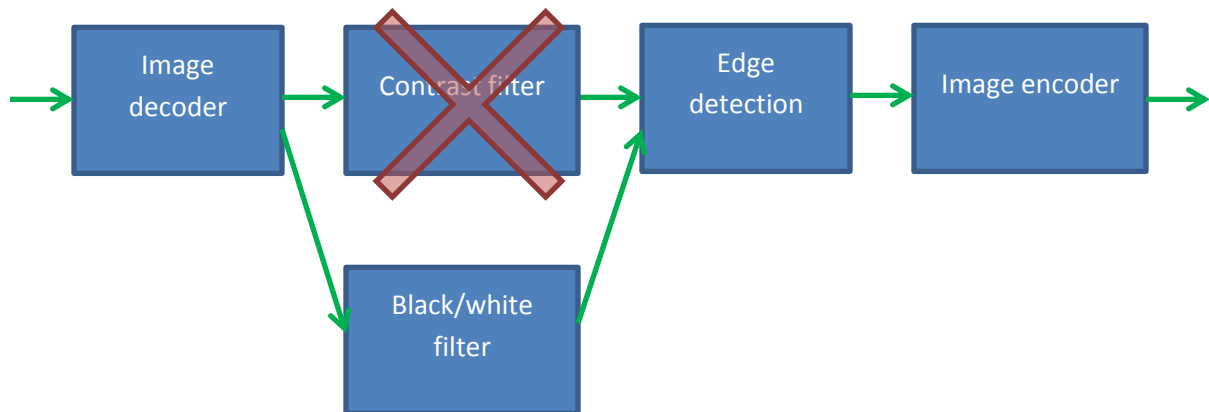
Een pipeline wordt toegepast als er verschillende acties, of filters achter elkaar moeten gebeuren. Het is dus een rij logische blokken. Deze blokken worden nodes genoemd.

Hieronder is een voorbeeld te zien van een image-processing pipeline, elk node doet zijn eigen werk, niet wetende wat de rest van de pipeline doet, en het resultaat wordt braaf doorgegeven aan de volgende node. Het werk dat zo'n node precies doet kan van alles zijn. Van een simpel optel sommetje tot complexe beeldbewerking.

De groene pijltjes geven de flow van data weer. De data die in een node gaat is input-data, de data die uit een node komt, output data.



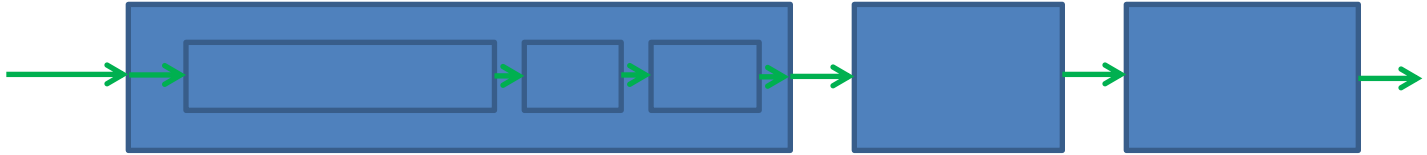
Eén van de grote voordelen van een pipeline is dat er heel makkelijk nodes kunnen worden veranderd, zonder dat dit andere nodes beïnvloedt. Hieronder is in het plaatje te zien hoe de contrast filter uit het voorbeeld hierboven vervangen wordt door een zwart/wit filter. Het eind resultaat van de pipeline zal hierdoor veranderen, maar de andere nodes worden hierdoor niet beïnvloed, en kunnen gewoon rustig hun eigen werk voortzetten.



In de volgende hoofdstukken staat wat uitgebreider uitgelegd hoe een pipeline precies werkt, hoe deze gebruikt en toegepast kan worden, wat de voor- en nadelen van een pipeline zijn, en hoe Philips gebruik maakt van de pipeline.

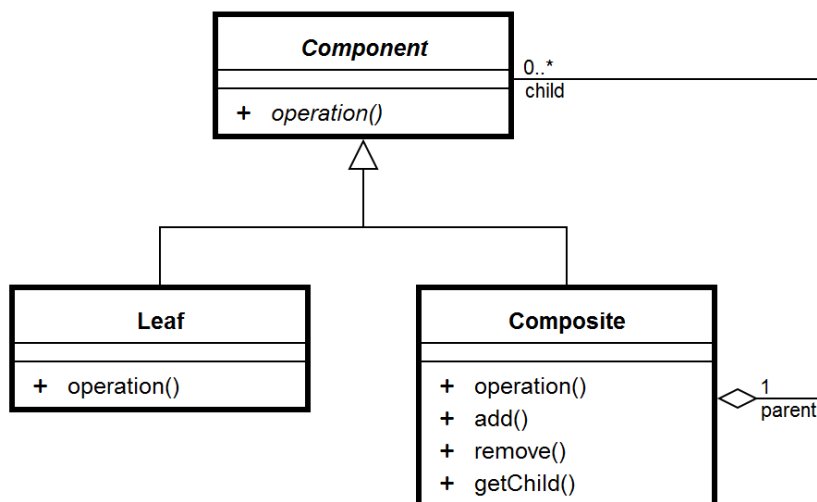
2. Composite patroon

Een belangrijk voordeel van een pipeline is dat het het composite patroon kan implementeren. Het idee van het composite patroon is dat een programma geen verschil hoeft te zien tussen een enkel object, en samengevoegde objecten. In het plaatje hieronder is geprobeerd dit duidelijk te maken.



Zoals te zien is in dit plaatje, bevat de eerste node in deze pipeline weer een hele pipeline. Het programma maakt dus geen onderscheid tussen één enkele node, en een verzameling van samengevoegde nodes. Een pipeline, dat eigenlijk niet meer is dan een verzameling nodes, wordt op zichzelf weer gezien als een node.

Het UML diagram van het composite patroon is hieronder weergegeven.



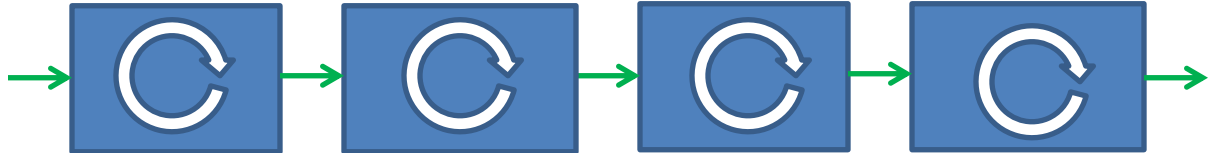
Zoals je hier kunt zien, erft de composite van de interface component, en beheert de composite ook een collectie met componenten. Dit houdt in dat composite naast een collectie van het object leaf (wat ook van component erft), ook andere composite objecten kan beheeren. Dit is precies hetzelfde als dat een pipeline behalve een bepaalde type pipeline node, ook een andere pipeline zou kunnen bevatten.

Om dit mogelijk te maken, moet de pipeline zelf een instantie zijn van pipeline node. Hoe dit er in code vorm uit ziet zal in het hoofdstuk “implementatie” duidelijk worden gemaakt.

3. Multithreading

Behalve dat nodes binnen een pipeline gemakkelijk kunnen worden vervangen door andere nodes, brengt het feit dat nodes onafhankelijk van elkaar functioneren nog een belangrijk voordeel met zich mee. Zo is het binnen een pipeline heel erg simpel om binnen een node logica uit te voeren op een andere thread. Sterker nog, als de pipeline goed gebruikt wordt, is dit zelfs noodzakelijk. Waarom dit noodzakelijk is wordt verder uitgelegd in het volgende hoofdstuk “queues”. Nu gaan we in op de voordelen die wat meer voor de hand liggen.

Een pipeline waar elke node zijn logica uitvoert op een eigen thread, kan als volgt worden weergegeven:



De witte pijlen stellen hier de threads voor.

Als er geen verschillende threads zouden worden gebruikt per node, moet het hele programma als nog wachten tot één node klaar is met zijn taak, voor de volgende taak kan worden uitgevoerd. Als alle nodes een eigen thread hebben, kan de rest van het programma verder gaan met andere taken terwijl de pipeline nodes het rekenwerk op een andere thread uitvoeren.

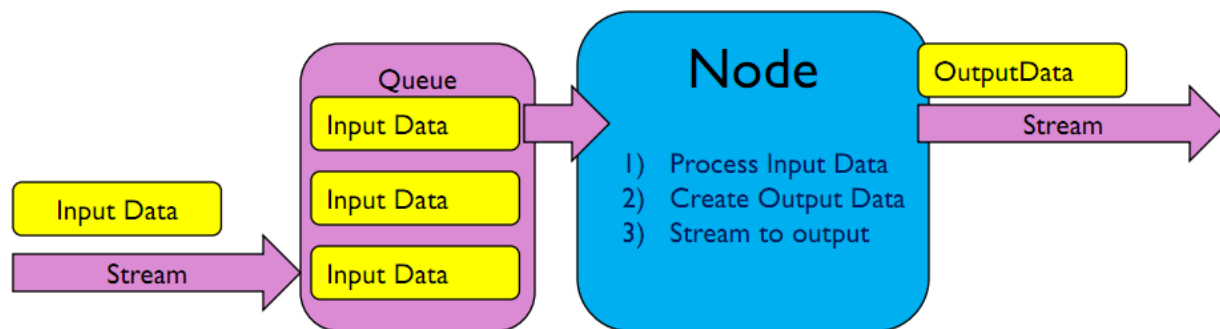
Een nadeel waar je dan meteen tegenaan loopt, is het feit dat, als de pipeline nodes achter elkaar hun werk moeten uitvoeren, en er is een node af en toe iets langzamer dan de ander zijn, er een opstopping kan voorkomen. Omdat de nodes achter zo’n node die een keer iets langer bezig is, allemaal hun data niet kwijt kunnen. Ook voor dit probleem is een oplossing, deze staat beschreven in het volgende hoofdstuk.

4. Queue

Zoals in het vorige hoofdstuk omschreven staat kan het nog steeds voorkomen dat sommige nodes zwaardere taken hebben dan anderen. Deze nodes zijn dan langer bezig en kunnen een opstopping veroorzaken.

Een oplossing hiervan is het gebruik maken van een queue (rij). Als een node nog bezig is met zijn taak, maar er komt wel al nieuwe data, dan wordt deze in de rij geplaatst. Wanneer de node dan klaar is met zijn taak, pakt hij de volgende data die in de rij staat, en gaat daar aan werken.

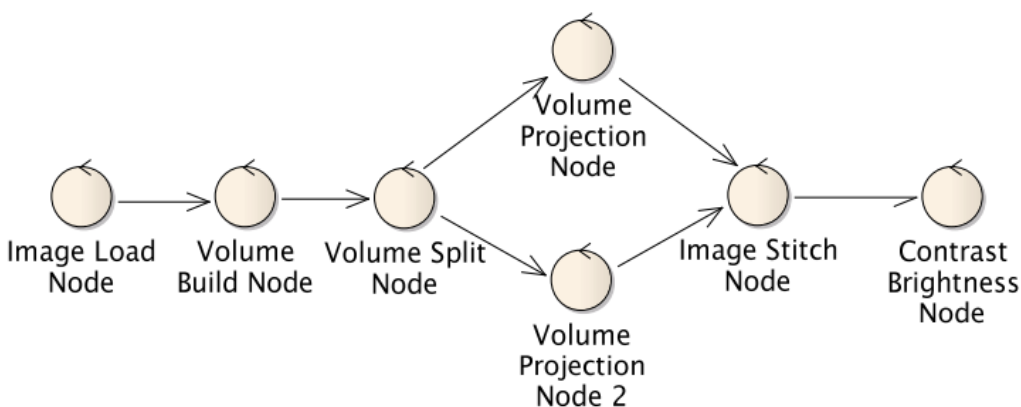
Hieronder is afgebeeld hoe dit er ongeveer uit ziet.



De Input-data gaat de queue in, en de node pakt een nieuw stuk data uit de queue wanneer deze klaar is met de huidige taak.

Belangrijk hierbij is wel dat de nodes alsnog het beste even lang over het verwerken van data kunnen doen. Als er een node aan het einde twee maal zo lang doet over zijn taak dan de voorliggende nodes, zal de queue vol blijven lopen. Er zijn wat trucjes die kunnen worden toegepast in een zo'n geval. Zo kan een node die te lang over zijn taak doet bijvoorbeeld worden opgedeeld in twee nodes die allebei een deel doet van de oorspronkelijke taak. Of de data kan worden opgedeeld zodat er twee verschillende nodes met dezelfde logica aan hetzelfde probleem kunnen werken.

In de illustratie is te zien hoe dit in zijn werk zal gaan.



4.1 Priority

Het kan voorkomen dat sommige stukken data belangrijker zijn dan andere. In dit geval wil je dat de belangrijke stukken data eerder door de pipeline gaan, en dus voorrang krijgen op de minder belangrijke data.

Om dit in de praktijk voor te stellen moet je bijvoorbeeld denken aan een programma dat een plaatje van hoge moet weergeven. In dit programma is niet het hele plaatje te zien, maar slechts een deel. Toch gaat het programma alvast berekenen hoe de rest van het plaatje moet worden weergegeven, zodat als de gebruiker hier naartoe scrollt, hij niet zo lang hoeft te wachten. Maar stel dat het programma gaat rekenen aan de rechterkant van het plaatje, en de gebruiker scrollt naar beneden, dan is het wenselijk dat het programma zo snel mogelijk toch gaat berekenen wat de gebruiker wil zien. De berekening voor het plaatje beneden is dus belangrijker dan de berekening voor de rechterkant van het plaatje, dit moet dus voorrang krijgen.

Om dit te implementeren kan bijvoorbeeld gebruik gemaakt worden van twee queues. De ene queue is voor hoge prioriteit opdrachten, en de andere queue voor opdrachten die minder belangrijk zijn. Wanneer een node klaar is met rekenen aan zijn taak, kijkt deze eerst of er data in de queue met hoge prioriteit ligt, en gaat pas in de lage prioriteit queue kijken als deze rij leeg is. Op die manier krijgt data met een hoge prioriteit altijd voorrang.

4.2 Throttling

Het kan zijn dat een rij toch langzaam vol loopt. Om hier toch controle over te houden, wordt throttling toegepast. Met throttling doe je eigenlijk niet meer dan het bepalen van de grote van een rij. Als de queue vol zit, dan mag er geen data meer binnenkomen.

Wat er met deze data gebeurt hangt van het doel van de pipeline af. Als alle data berekend moet worden, en er mag niks verloren gaan, maar tijd maakt niets uit, dan zal de pipeline moeten wachten tot de node waarvan de queue vol zat weer klaar is, en nieuwe data mag ontvangen. Maakt tijd wel uit, en is het belangrijker dat er real time data uit de pipeline komt dan dat alle data aankomt, dan kan er voor gekozen worden om data dat niet meer in een queue past weg te gooien.

Throttling kent twee extremen. Zo kan de limit van een queue 1 zijn, wat er voor zorgt dat de pipeline voorspellend is, maar wel blokkeert als een node niet op tijd klaar is. Maar de limit kan ook oneindig zijn. Dit zorgt ervoor dat de pipeline nooit blokkeert, maar dat het heel lang kan duren voordat data aankomt, en dat het ook niet goed te voorspellen is hoe lang dit gaat duren.

5. Processing time

De processing time van een (simpele) pipeline is de tijd die data erover doet om vanaf het moment dat het in de pipeline gestopt wordt, naar de andere kant te gaan. Bij een pipeline die gebruik maakt van queues is dit veel moeilijker uit te rekenen, of soms helemaal niet (als er bijvoorbeeld data weggegooid wordt in geval van een volle queue, of als queues oneindig groot kunnen worden). Als voorbeeld gebruiken we een simpele, blokkerende pipeline.

De tijd die data er over doet om van het begin naar het einde van de pipeline te gaan, is uit te rekenen door de tijd van elke node bij elkaar op te tellen. De tijd die data erover doet om door de onderstaande pipeline te gaan, is dus $1 + 1 + 1 = 3$ seconden.



Een belangrijk voordeel van een pipeline is, dat een pipeline verschillende stukken data tegelijk kan verwerken. Als de eerste node van de pipeline klaar is met de data, en zijn data doorgeeft aan de volgende node, kan er wel al nieuwe data de pipeline in. Dit betekent dat bovenstaande pipeline 3 seconden over een stuk data doet, maar wel elke seconde nieuwe data kan accepteren, en outputten.

Hoe kleiner nodes gemaakt worden, hoe meer data er dus tegelijkertijd berekend kan worden, en hoe sneller de dataflow zal worden. Ter vergelijking: nu kunnen er in 5 seconden 3 stukken data helemaal door de pipeline gaan, maar als de nodes zouden worden samengevoegd, zouden dezelfde 3 stukken data er 9 seconden over doen.

6. Distributed computing

6.1 Inleiding

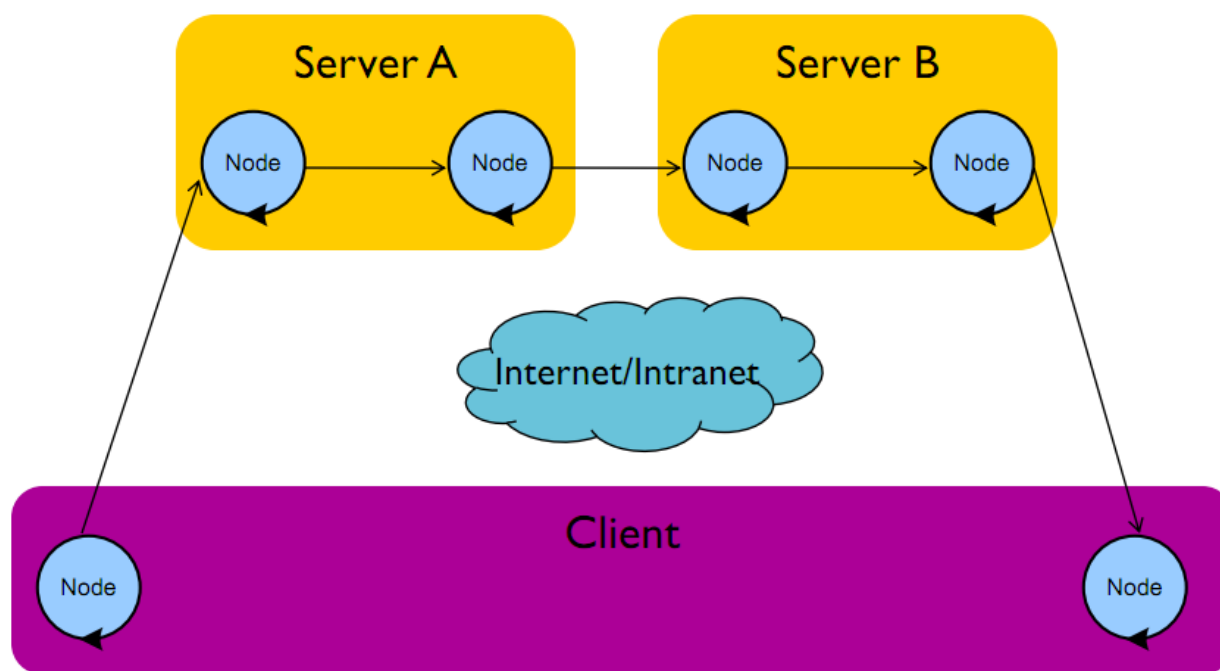
Distributed computing is ook een architectuur pattern. Dit patroon wordt verder uitgelegd in het hoofdstuk distributed computing.

In dit deel van het verslag wordt behandeld hoe distributed computing toe te passen is bij een pipeline, en hoe dit in zijn werk gaat.

6.2 Pipeline en distributed computing

Het principe van het gebruiken van distributed computing voor een pipeline ligt erg voor de hand. Aangezien nodes heel makkelijk uit elkaar kunnen worden gehaald, en zelf niet bewust hoeven te zijn van hun bijdrage in het grote geheel, is het ook erg makkelijk om een node bijvoorbeeld op een andere computer te laten draaien. Zo kan een pipeline makkelijk worden verdeeld over meerdere computers.

In de afbeelding hieronder is weergegeven hoe dit eruit ziet.

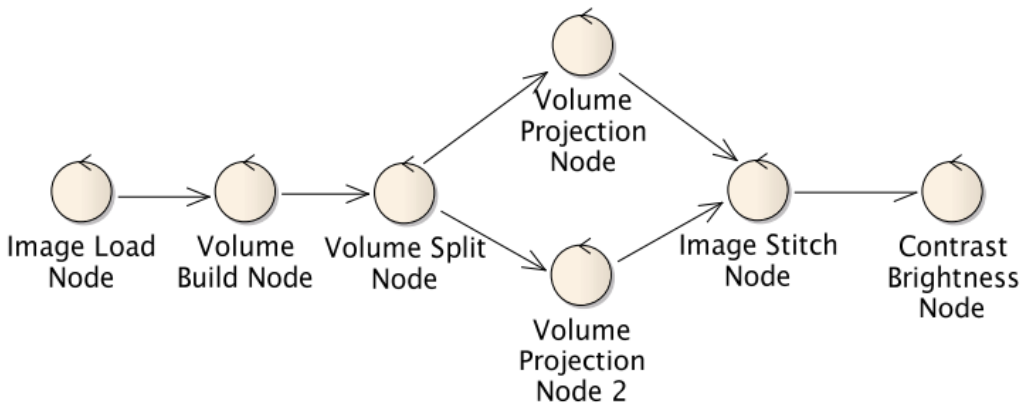


De data gaat oorspronkelijk bij de client de pipeline in. Maar zodra deze uit de eerste node komt, wordt deze naar server A gestuurd. Deze zal, samen met server B, wat andere berekeningen uitvoeren. Uiteindelijk wordt het resultaat weer naar de client gestuurd, die wat met de data kan doen. De client hoeft op deze manier niet veel rekenkracht te hebben.

Een bijkomend voordeel is dat er op deze manier ook op een slimme manier omgegaan kan worden met de hardware die gebruikt wordt voor de pipeline. Omdat specifieke nodes vaak één taak hebben, kan er voor bepaalde nodes hardware speciaal worden aangeschaft voor de taak die ze uitvoeren.

7. Meerdere inputs/outputs

In de vorige paar hoofdstukken was een node een simpel blokje met één input en één output. Dit is niet altijd het geval. Het is gewoon toegestaan om een node meerdere inputs of meerdere outputs te geven. Dit kan bijvoorbeeld als voordeel bieden dat data kan worden gesplitst en weer kan worden samengevoegd, zoals eerder in hoofdstuk “queue” besproken is.



De volume Split Node heeft twee outputs, en wordt gebruikt om een stuk data op te splitsen. Deze data kan vervolgens verwerkt worden waarna deze naar de Image Stitch Node worden gestuurd. Deze node heeft twee inputs, en wordt gebruikt om de data weer te combineren tot één dataobject.

Het gebruik van meerdere inputs en outputs kan voor problemen zorgen. Als voorbeeld nemen we de afbeelding hierboven. Stel dat Volume Projection Node drie keer zo snel werkt als Volume Projection Node 2. De image stitch node krijgt dan misschien niet de juiste twee halve frames binnen om aan elkaar te maken. Zelfs als beide nodes even snel zijn kan dit voorkomen als er één een tijdje vastloopt. Om dit te voorkomen kan er gebruik worden gemaakt van id's. Elk data pakket krijgt een nummer mee. Wanneer een frame wordt opgesplitst, krijgen beide pakketten hetzelfde nummer. De image stitch node kan vervolgens checken of de binnengekomen pakketten hetzelfde nummer hebben, en als dit niet het geval is wachten (bijvoorbeeld met het gebruik van een queue) op een pakket dat wel bij het andere pakket past.

Verder is het aan de nodes zelf om te beslissen hoe ze data op binnenkomende pinnen afhandelen en hoe ze data verzenden. Zo kan een node wachten tot er nieuwe data op alle pinnen is binnen gekomen, maar ook besluiten om een berekening uit te voeren als er maar één input verandert.

8. Conclusie

Ter conclusie van het pipeline patroon nog even de behandelde voor- en nadelen van het gebruik van een pipeline op een rijtje.

8.1 Voordelen

- Nodes zijn makkelijk te verwisselen .
- Een pipeline heeft geen loopjes, data kan niet oneindig blijven rondcirculeren.
- Een pipeline is makkelijk te verdelen over meerdere processen/computers.
- Een pipeline is een composite, en kan daarom zelf ook als node dienen voor een andere pipeline.
- Een pipeline kan meerdere taken tegelijk doen (wel in verschillende nodes).
- Een pipeline kan prioriteit geven aan bepaalde data.

8.2 Nadelen

Eén node kan de hele pipeline blokkeren.